



Open CASCADE Technology
6.9.0

OCAF

May 8, 2015

Contents

1	Introduction	1
2	Basic Concepts	3
2.1	Overview	3
2.2	Applications and documents	3
2.3	The document and the data framework	4
2.3.1	Documents	6
2.3.2	Shape attribute	7
2.3.3	Standard attributes	7
2.3.4	Visualization attributes	7
2.3.5	Function services	7
3	Data Framework Services	9
3.1	Overview	9
3.2	The Tag	10
3.2.1	Creating child labels using random delivery of tags	11
3.2.2	Creation of a child label by user delivery from a tag	11
3.3	The Label	11
3.3.1	Label creation	12
3.3.2	Creating child labels	12
3.3.3	Retrieving child labels	12
3.3.4	Retrieving the father label	13
3.4	The Attribute	13
3.4.1	Retrieving an attribute from a label	13
3.4.2	Identifying an attribute using a GUID	14
3.4.3	Attaching an attribute to a label	14
3.4.4	Testing the attachment to a label	14
3.4.5	Removing an attribute from a label	14
3.4.6	Specific attribute creation	14
4	Standard Document Services	18
4.1	Overview	18
4.2	The Application	18
4.2.1	Creating an application	18
4.2.2	Creating a new document	18
4.2.3	Retrieving the application to which the document belongs	18
4.3	The Document	18
4.3.1	Accessing the main label of the framework	19
4.3.2	Retrieving the document from a label in its framework	19

4.3.3	Saving the document	19
4.3.4	Opening the document from a file	20
4.3.5	Cutting, copying and pasting inside a document	20
4.4	External Links	21
4.4.1	Copying the document	21
5	OCAF Shape Attributes	23
5.1	Overview	23
5.2	Services provided	26
5.2.1	Registering shapes and their evolution	26
5.2.2	Using naming resources	26
5.2.3	Reading the contents of a named shape attribute	26
5.2.4	Selection Mechanism	27
5.2.5	Exploring shape evolution	27
6	Standard Attributes	28
6.1	Overview	28
6.2	Services common to all attributes	29
6.2.1	Accessing GUIDs	29
6.2.2	Conventional Interface of Standard Attributes	30
7	Visualization Attributes	31
7.1	Overview	31
7.2	Services provided	31
7.2.1	Defining an interactive viewer attribute	31
7.3	Defining a presentation attribute	31
7.3.1	Creating your own driver	31
7.3.2	Using a container for drivers	31
8	Function Services	33
8.1	Finding functions, their owners and roots	33
8.2	Storing and accessing information about function status	33
8.3	Propagating modifications	33
9	XML Support	36
9.1	Document Drivers	36
9.2	Attribute Drivers	37
9.3	XML Document Structure	37
9.4	XML Schema	39
10	GLOSSARY	40
11	Samples	42

11.1 Implementation of Attribute Transformation in a CDL file	42
11.2 Implementation of Attribute Transformation in a CPP file	43
11.3 Implementation of typical actions with standard OCAF attributes.	47

1 Introduction

This manual explains how to use the Open CASCADE Application Framework (OCAF). It provides basic documentation on using OCAF. For advanced information on OCAF and its applications, see our offerings on our web site at www.opencascade.org/support/training/

OCAF (the Open CASCADE Application Framework) is a RAD (Rapid Application Development) framework used for specifying and organizing application data. To do this, OCAF provides:

- Ready-to-use data common to most CAD/CAM applications,
- A scalable extension protocol for implementing new application specific data,
- An infrastructure
- To attach any data to any topological element
- To link data produced by different applications (*associativity of data*)
- To register the modeling process - the creation history, or parametrics, used to carry out the modifications.

Using OCAF, the application designer concentrates on the functionality and its specific algorithms. In this way, he avoids architectural problems notably implementing Undo-redo and saving application data.

In OCAF, all of the above are already handled for the application designer, allowing him to reach a significant increase in productivity.

In this respect, OCAF is much more than just one toolkit among many in the CAS.CADE Object Libraries. Since it can handle any data and algorithms in these libraries - be it modeling algorithms, topology or geometry - OCAF is a logical supplement to these libraries.

The table below contrasts the design of a modeling application using object libraries alone and using OCAF.

Table 1: Services provided by OCAF

Development tasks	Comments	Without OCAF	With OCAF
Creation of geometry	Algorithm Calling the modeling libraries	To be created by the user	To be created by the user
Data organization	Including specific attributes and modeling process	To be created by the user	Simplified
Saving data in a file	Notion of document	To be created by the user	Provided
Document-view management		To be created by the user	Provided
Application infrastructure	New, Open, Close, Save and Save As File menus	To be created by the user	Provided
Undo-Redo	Robust, multi-level	To be created by the user	Provided
Application-specific dialog boxes		To be created by the user	To be created by the user

The relationship between OCAF and the Open CASCADE Technology (OCCT) Object Libraries can be seen in the image below.

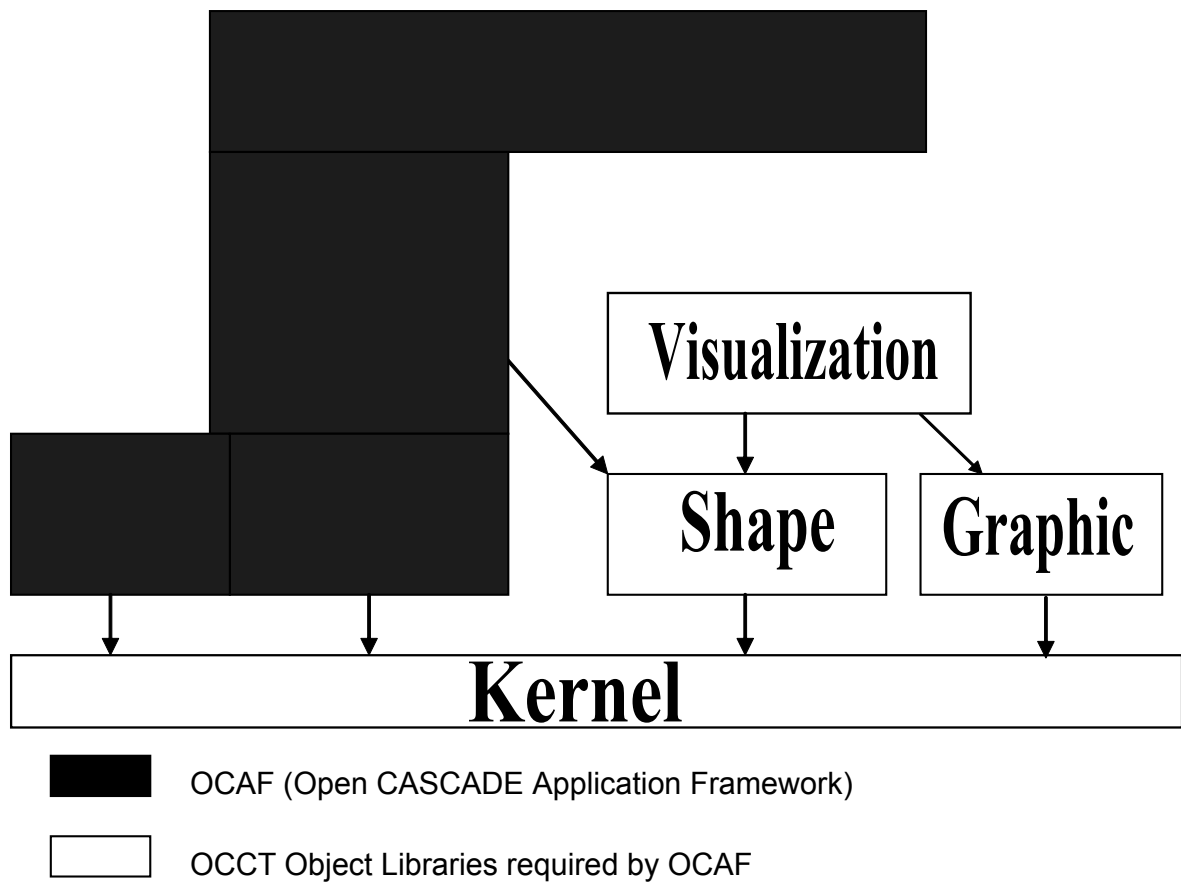


Figure 1: OCAF Architecture

In the image, the OCAF (Open CASCADE Application Framework) is shown with black rectangles and OCCT Object Libraries required by OCAF are shown with white rectangles.

The subsequent chapters of this document explain the concepts and show how to use the services of OCAF.

2 Basic Concepts

2.1 Overview

In most existing geometric modeling systems, the data structure is shape driven. They usually use a brep model, where solids and surfaces are defined by a collection of entities such as faces, edges etc., and by attributes such as application data. These attributes are attached to the entities. Examples of application data include:

- color,
- material,
- information that a particular edge is blended.

A shape, however, is inevitably tied to its underlying geometry. And geometry is highly subject to change in applications such as parametric modeling or product development. In this sort of application, using a brep (boundary representation) data structure proves to be a not very effective solution. A solution other than the shape must be found, i.e. a solution where attributes are attached to a deeper invariant structure of the model. Here, the topology itself will be one attribute among many.

In OCAF, data structure is reference key-driven. The reference key is implemented in the form of labels. Application data is attached to these labels as attributes. By means of these labels and a tree structure they are organized in, the reference key aggregates all user data, not just shapes and their geometry. These attributes have similar importance; no attribute is master in respect of the others.

The reference keys of a model - in the form of labels - have to be kept together in a single container. This container is called a document.

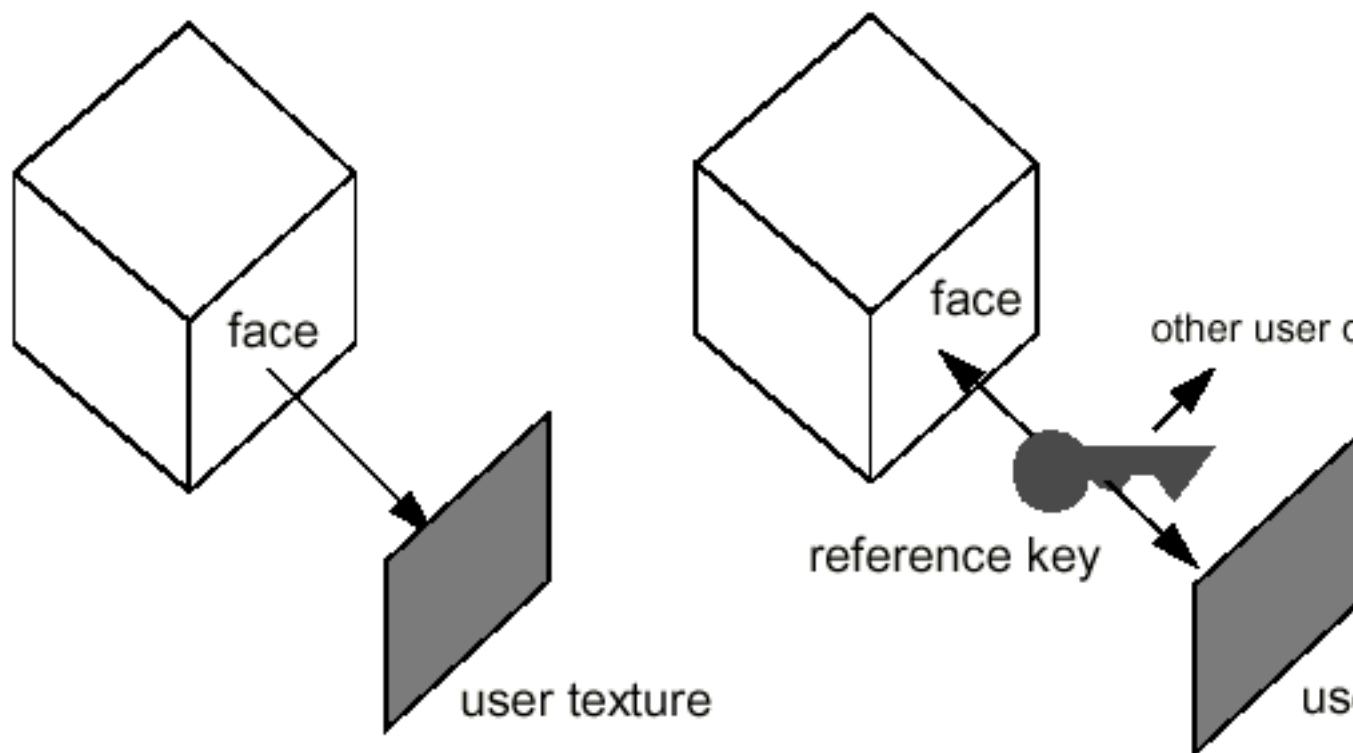


Figure 2: Topology-driven vs. reference key-driven approaches

2.2 Applications and documents

OCAF documents are in turn managed by an OCAF application, which is in charge of:

- Creating new documents
- Saving documents and opening them
- Initializing document views.

Apart from their role as a container of application data, documents can refer to each other; Document A, for example, can refer to a specific label in Document B. This functionality is made possible by means of the reference key.

2.3 The document and the data framework

Inside a document, there is a data framework, a model, for example. This is a set of labels organized in a tree structure characterized by the following features:

- The first label in a framework is the root of the tree;
- Each label has a tag expressed as an integer value;
- Sub-labels of a label are called its children;
- Each label which is not the root has one father – label from an upper level of the framework;
- Labels which have the same father are called brothers;
- Brothers cannot share the same tag;
- A label is uniquely defined by an entry expressed as a list of tags (entry) of fathers from the root: this list of tags is written from right to left: tag of label, tag of its father, tag of father of its father,..., 0 (tag of the root label).

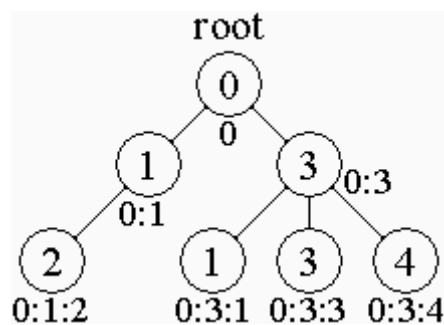


Figure 3: A simple framework model

In the above figure inside the circles are the tags of corresponding labels. Under the circles are the lists of tags. The root label always has a zero tag.

The children of a root label are middle-level labels with tags 1 and 3. These labels are brothers.

List of tags of the right-bottom label is "0:3:4": this label has tag 4, its father (with entry "0:3") has tag 3, father of father has tag 0 (the root label always has "0" entry).

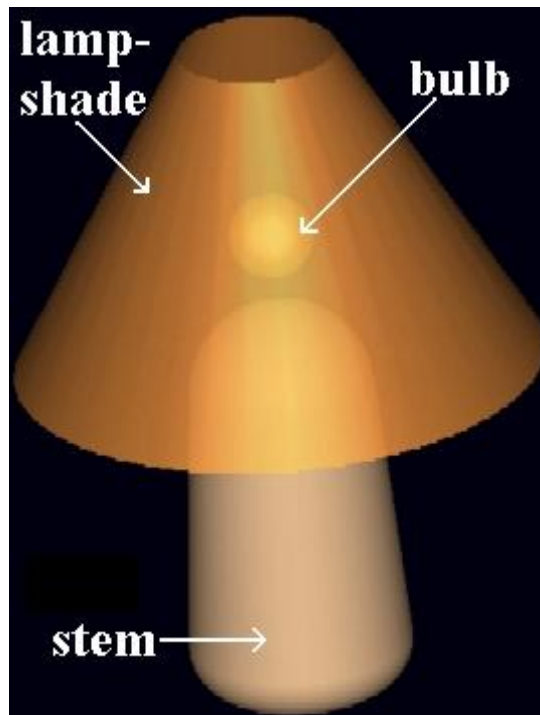
For example, an application for designing table lamps will first allocate a label for the lamp unit (the lamp is illustrated below). The root label never has brother labels, so, for a lot of lamps in the framework allocation, one of the root label sub-labels for the lamp unit is used. By doing so, you would avoid any confusion between table lamps in the data framework. Parts of the lamp have different material, color and other attributes, so, for each sub-unit of the lamp a child label of the lamp label with specified tags is allocated:

- a lamp-shade label with tag 1
- a bulb label with tag 2

- a stem label with tag 3

Label tags are chosen at will. They are just identifiers of the lamp parts. Now you can refine all units: set to the specified label geometry, color, material and other information about the lamp or it's parts. This information is placed into special attributes of the label: the pure label contains no data – it is only a key to access data.

The thing to remember is that tags are private addresses without any meaning outside the data framework. It would, for instance, be an error to use part names as tags. These might change or be removed from production in next versions of the application, whereas the exact form of that part might be what you wanted to use in your design, the part name could be integrated into the framework as an attribute.



So, after the user changes the lamp design, only corresponding attributes are changed, but the label structure is maintained. The lamp shape must be recreated by new attribute values and attributes of the lamp shape must refer to a new shape.

2.3.2 Shape attribute

The shape attribute implements the functionality of the OCCT topology manipulation:

- reference to the shapes
- tracking of shape evolution

2.3.3 Standard attributes

Several ready-to-use base attributes already exist. These allow operating with simple common data in the data framework (for example: integer, real, string, array kinds of data), realize auxiliary functions (for example: tag sources attribute for the children of the label counter), create dependencies (for example: reference, tree node)....

2.3.4 Visualization attributes

These attributes allow placing viewer information to the data framework, visual representation of objects and other auxiliary visual information, which is needed for graphical data representation.

2.3.5 Function services

Where the document manages the notification of changes, a function manages propagation of these changes. The function mechanism provides links between functions and calls to various algorithms.

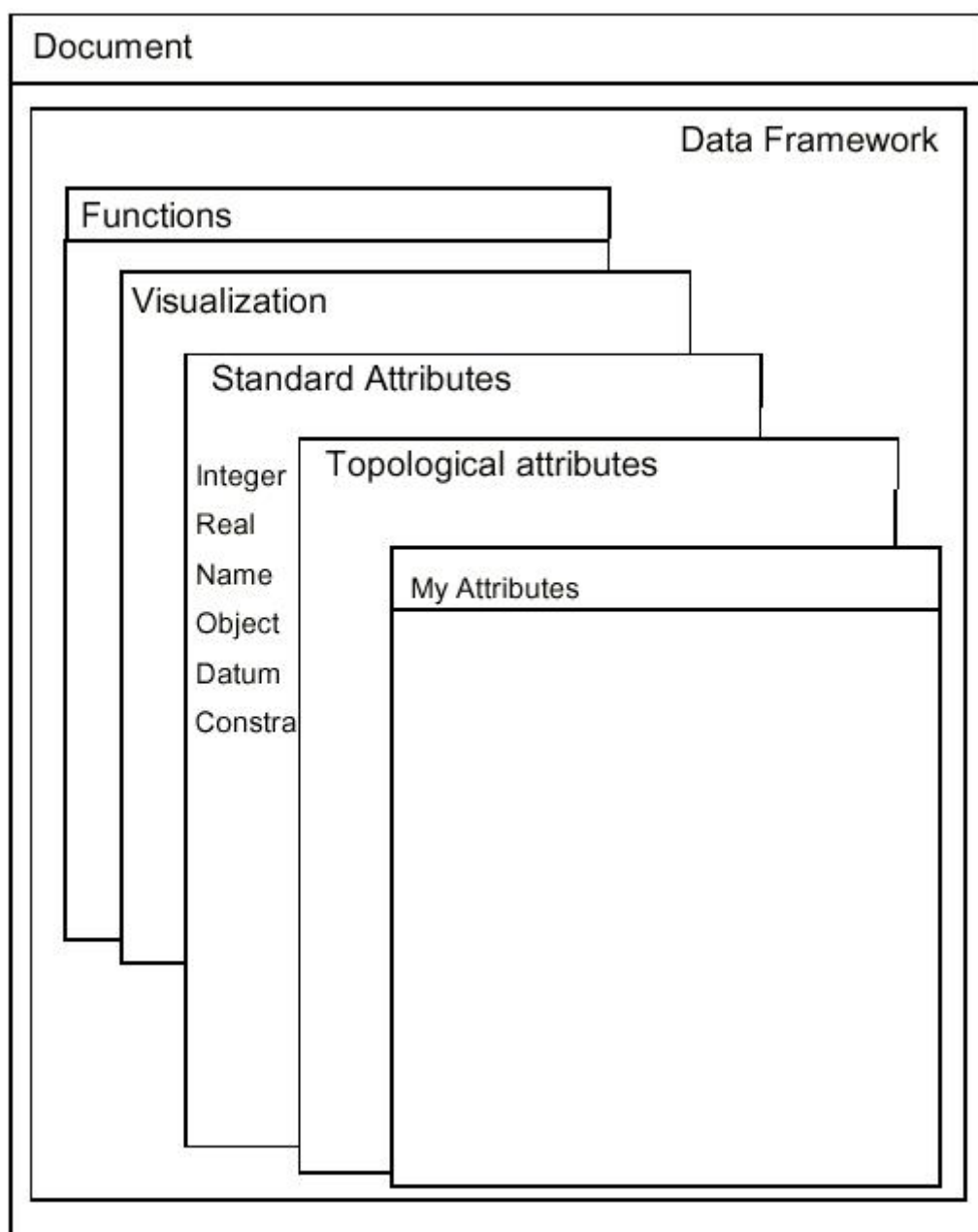


Figure 4: Document structure

3 Data Framework Services

3.1 Overview

The data framework offers a single environment in which data from different application components can be handled.

This allows you to exchange and modify data simply, consistently, with a maximum level of information, and with stable semantics.

The building blocks of this approach are:

- The tag
- The label
- The attribute

As it has been mentioned earlier, the first label in a framework is the root label of the tree. Each label has a tag expressed as an integer value, and a label is uniquely defined by an entry expressed as a list of tags from the root, 0:1:2:1, for example.

Each label can have a list of attributes, which contain data, and several attributes can be attached to a label. Each attribute is identified by a GUID, and although a label may have several attributes attached to it, it must not have more than one attribute of a single GUID.

The sub-labels of a label are called its children. Conversely, each label, which is not the root, has a father. Brother labels cannot share the same tag.

The most important property is that a label's entry is its persistent address in the data framework.

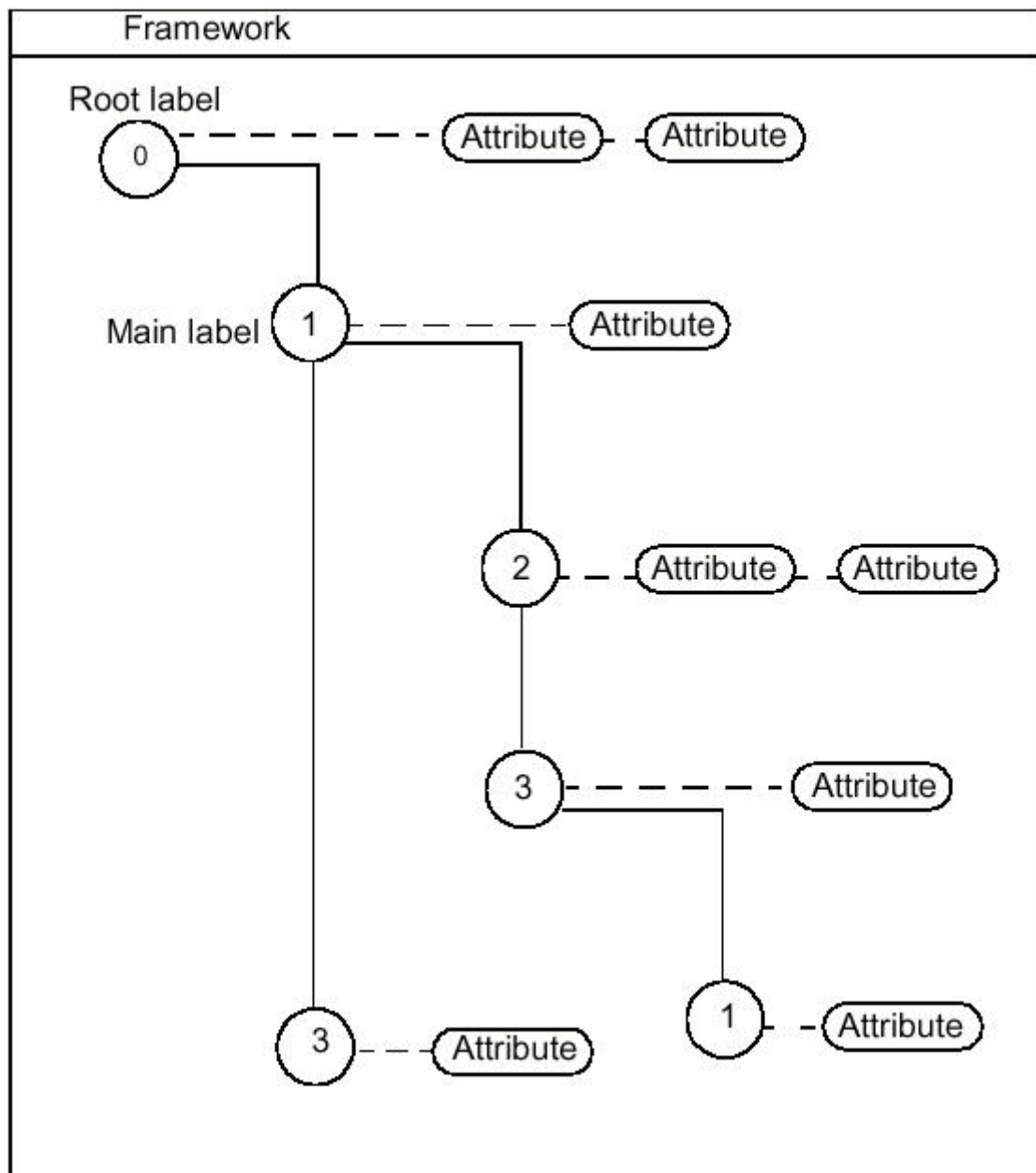


Figure 5: Contents of a document

3.2 The Tag

A tag is an integer, which identifies a label in two ways:

- Relative identification
- Absolute identification.

In relative identification, a label's tag has a meaning relative to the father label only. For a specific label, you might, for example, have four child labels identified by the tags 2, 7, 18, 100. In using relative identification, you ensure that you have a safe scope for setting attributes.

In absolute identification, a label's place in the data framework is specified unambiguously by a colon-separated list of tags of all the labels from the one in question to the root of the data framework. This list is called an entry. *TDF_Tool::TagList* allows retrieving the entry for a specific label.

In both relative and absolute identification, it is important to remember that the value of an integer has no intrinsic semantics whatsoever. In other words, the natural sequence that integers suggest, i.e. 0, 1, 2, 3, 4 ... - has no importance here. The integer value of a tag is simply a key.

The tag can be created in two ways:

- Random delivery
- User-defined delivery

As the names suggest, in random delivery, the tag value is generated by the system in a random manner. In user-defined delivery, you assign it by passing the tag as an argument to a method.

3.2.1 Creating child labels using random delivery of tags

To append and return a new child label, you use *TDF_TagSource::NewChild*. In the example below, the argument *level2*, which is passed to *NewChild*, is a *TDF_Label*.

```
TDF_Label child1 = TDF_TagSource::NewChild (level2);
TDF_Label child2 = TDF_TagSource::NewChild (level2);
```

3.2.2 Creation of a child label by user delivery from a tag

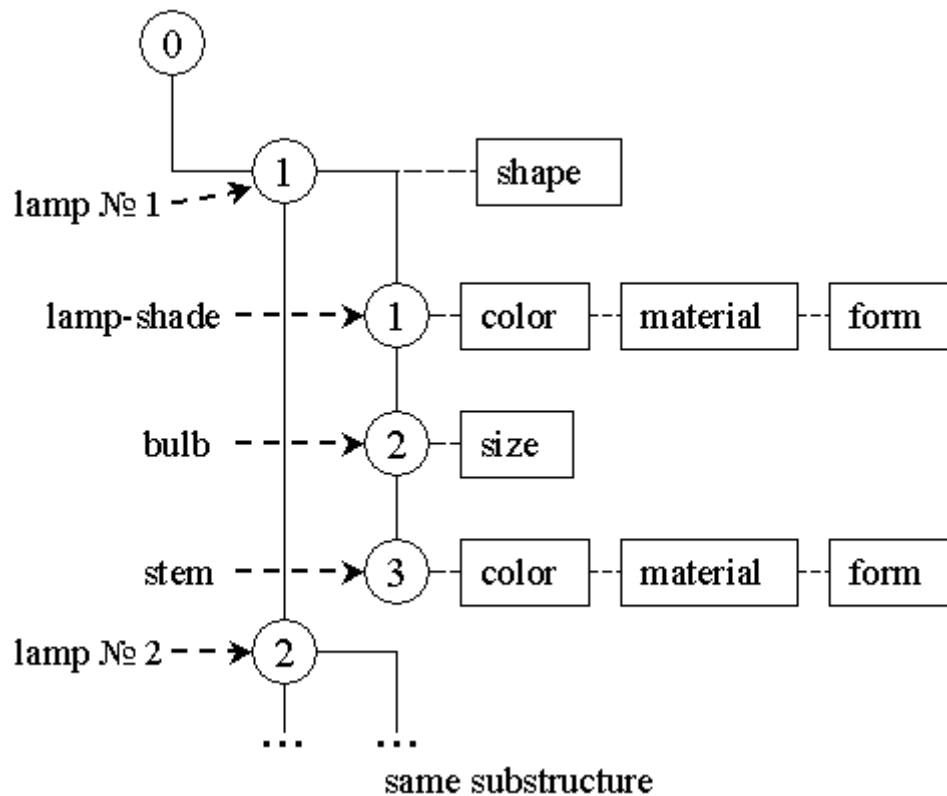
The other way to create a child label from a tag is by user delivery. In other words, you specify the tag, which you want your child label to have.

To retrieve a child label from a tag which you have specified yourself, you need to use *TDF_Label::FindChild* and *TDF_Label::Tag* as in the example below. Here, the integer 3 designates the tag of the label you are interested in, and the Boolean false is the value for the argument *create*. When this argument is set to *false*, no new child label is created.

```
TDF_Label achild = root.FindChild(3,Standard_False);
if (!achild.IsNull()) {
    Standard_Integer tag = achild.Tag();
}
```

3.3 The Label

The tag gives a persistent address to a label. The label – the semantics of the tag – is a place in the data framework where attributes, which contain data, are attached. The data framework is, in fact, a tree of labels with a root as the ultimate father label (refer to the following figure):



Label can not be deleted from the data framework, so, the structure of the data framework that has been created can not be removed while the document is opened. Hence any kind of reference to an existing label will be actual while an application is working with the document.

3.3.1 Label creation

Labels can be created on any labels, compared with brother labels and retrieved. You can also find their depth in the data framework (depth of the root label is 0, depth of child labels of the root is 1 and so on), whether they have children or not, relative placement of labels, data framework of this label. The class *TDF_Label* offers the above services.

3.3.2 Creating child labels

To create a new child label in the data framework using explicit delivery of tags, use *TDF_Label::FindChild*.

```
//creating a label with tag 10 at Root
TDF_Label lab1 = aDF->Root().FindChild(10);

//creating labels 7 and 2 on label 10
TDF_Label lab2 = lab1.FindChild(7);

TDF_Label lab3 = lab1.FindChild(2);
```

You could also use the same syntax but add the Boolean *true* as a value of the argument **create**. This ensures that a new child label will be created if none is found. Note that in the previous syntax, this was also the case since **create** is *true* by default.

```
TDF_Label level1 = root.FindChild(3,Standard_True);
TDF_Label level2 = level1.FindChild(1,Standard_True);
```

3.3.3 Retrieving child labels

You can retrieve child labels of your current label by iteration on the first level in the scope of this label.

```
TDF_Label current;
//
for (TDF_ChildIterator it1 (current,Standard_False); it1.More(); it1.Next()) {
  achild = it1.Value();
  //
  // do something on a child (level 1)
  //
}
```

You can also retrieve all child labels in every descendant generation of your current label by iteration on all levels in the scope of this label.

```
for (TDF_ChildIterator itall (current,Standard_True); itall.More(); itall.Next()) {
  achild = itall.Value();
  //
  // do something on a child (all levels)
  //
}
```

Using *TDF_Tool::Entry* with *TDF_ChildIterator* you can retrieve the entries of your current label's child labels as well.

```
void DumpChildren(const TDF_Label& aLabel)
{
  TDF_ChildIterator it;
  TCollection_AsciiString es;
  for (it.Initialize(aLabel,Standard_True); it.More(); it.Next()) {
    TDF_Tool::Entry(it.Value(),es);
    cout << es.ToCString() << endl;
  }
}
```

3.3.4 Retrieving the father label

Retrieving the father label of a current label.

```
TDF_Label father = achild.Father();
isroot = father.IsRoot();
```

3.4 The Attribute

The label itself contains no data. All data of any type whatsoever - application or non-application - is contained in attributes. These are attached to labels, and there are different types for different types of data. OCAF provides many ready-to-use standard attributes such as integer, real, constraint, axis and plane. There are also attributes for topological naming, functions and visualization. Each type of attribute is identified by a GUID.

The advantage of OCAF is that all of the above attribute types are handled in the same way. Whatever the attribute type is, you can create new instances of them, retrieve them, attach them to and remove them from labels, "forget" and "remember" the attributes of a particular label.

3.4.1 Retrieving an attribute from a label

To retrieve an attribute from a label, you use *TDF_Label::FindAttribute*. In the example below, the GUID for integer attributes, and *INT*, a handle to an attribute are passed as arguments to *FindAttribute* for the current label.

```
if (current.FindAttribute(TDataStd_Integer::GetID(),INT))
{
  // the attribute is found
}
else
{
  // the attribute is not found
}
```

3.4.2 Identifying an attribute using a GUID

You can create a new instance of an attribute and retrieve its GUID. In the example below, a new integer attribute is created, and its GUID is passed to the variable *guid* by the method *ID* inherited from *TDF_Attribute*.

```
Handle(TDataStd_Integer) INT = new TDataStd_Integer();
Standard_GUID guid = INT->ID();
```

3.4.3 Attaching an attribute to a label

To attach an attribute to a label, you use *TDF_Label::Add*. Repetition of this syntax raises an error message because there is already an attribute with the same GUID attached to the current label.

TDF_Attribute::Label for *INT* then returns the label *attach* to which *INT* is attached.

```
current.Add (INT); // INT is now attached to current
current.Add (INT); // causes failure
TDF_Label attach = INT->Label();
```

3.4.4 Testing the attachment to a label

You can test whether an attribute is attached to a label or not by using *TDF_Attribute::IsA* with the GUID of the attribute as an argument. In the example below, you test whether the current label has an integer attribute, and then, if that is so, how many attributes are attached to it. *TDataStd_Integer::GetID* provides the GUID argument needed by the method *IsAttribute*.

TDF_Attribute::HasAttribute tests whether there is an attached attribute, and *TDF_Tool::NbAttributes* returns the number of attributes attached to the label in question, e.g. *current*.

```
// Testing of attribute attachment
//
if (current.IsA(TDataStd_Integer::GetID())) {
// the label has an Integer attribute attached
}
if (current.HasAttribute()) {
// the label has at least one attribute attached
Standard_Integer nbatt = current.NbAttributes();
// the label has nbatt attributes attached
}
```

3.4.5 Removing an attribute from a label

To remove an attribute from a label, you use *TDF_Label::Forget* with the GUID of the deleted attribute. To remove all attributes of a label, *TDF_Label::ForgetAll*.

```
current.Forget(TDataStd_Integer::GetID());
// integer attribute is now not attached to current label
current.ForgetAll();
// current has now 0 attributes attached
```

3.4.6 Specific attribute creation

If the set of existing and ready to use attributes implementing standard data types does not cover the needs of a specific data presentation task, the user can build his own data type and the corresponding new specific attribute implementing this new data type.

There are two ways to implement a new data type: create a new attribute (standard approach), or use the notion of User Attribute by means of a combination of standard attributes (alternative way)

In order to create a new attribute in the standard way do the following:

- Create a class inherited from *TDF_Attribute* and implement all purely virtual and necessary virtual methods:

- **ID()** – returns a unique GUID of a given attribute
 - **Restore(attribute)** – sets fields of this attribute equal to the fields of a given attribute of the same type
 - **Paste(attribute, relocation_table)** – sets fields of a given attribute equal to the field values of this attribute ; if the attribute has references to some objects of the data framework and *relocation_table* has this element, then the given attribute must also refer to this object .
 - **NewEmpty()** - returns a new attribute of this class with empty fields
 - **Dump(stream)** - outputs information about a given attribute to a given stream debug (usually outputs an attribute of type string only)
- Create the persistence classes for this attribute according to the file format chosen for the document (see below).

Methods *NewEmpty*, *Restore* and *Paste* are used for the common transactions mechanism (Undo/Redo commands). If you don't need this attribute to react to undo/redo commands, you can write only stubs of these methods, else you must call the Backup method of the *TDF_Attribute* class every time attribute fields are changed.

If you use a standard file format and you want your new attributes to be stored during document saving and retrieved to the data framework whenever a document is opened, you must do the following:

1. If you place an attribute to a new package, it is desirable (although not mandatory) if your package name starts with letter "T" (transient), for example: attribute *TMyAttributePackage_MyAttribute* in the package *TMyAttributePackage*.
2. Create a new package with name "P[package name]" (for example *PMyAttributePackage*) with class *PMyAttributePackage_MyAttribute* inside. The new class inherits the *PDF_Attribute* class and contains fields of attributes, which must be saved or retrieved ("P" - persistent).
3. Create a new package with name "M[package name]" (for example *MMyAttributePackage*) with classes *MMyAttributePackage_MyAttributeRetrievalDriver* and *MMyAttributePackage_MyAttributeStorageDriver* inside. The new classes inherit *MDF_ARDriver* and *MDF_ASDriver* classes respectively and contain the translation functionality: from T... attribute to P... and vice versa (M - middle) (see the realization of the standard attributes).
4. M... package must contain *AddStorageDrivers(aDriverSeq : ASDriverHSequence from MDF)* and *AddRetrievalDrivers(aDriverSeq : ASDriverHSequence from MDF)* methods, which append to the given sequence of drivers *aDriverSeq*, which is a sequence of all new attribute drivers (see the previous point) used for the storage/retrieval of attributes. 5 Use the standard schema (*StdSchema* unit) or create a new one to add your P-package and compile it.

If you use the XML format, do the following:

1. Create a new package with the name *Xml[package name]* (for example *XmlMyAttributePackage*) containing class *XmlMyAttributePackage_MyAttributeDriver*. The new class inherits *XmlMDF_ADriver* class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages *XmlMDataStd*, for example). Add package method *AddDrivers* which adds your class to a driver table (see below).
2. Create a new package (or do it in the current one) with two package methods:
 - *Factory*, which loads the document storage and retrieval drivers; and
 - *AttributeDrivers*, which calls the methods *AddDrivers* for all packages responsible for persistence of the document.
3. Create a plug-in implemented as an executable (see example *XmlPlugin*). It calls a macro *PLUGIN* with the package name where you implemented the method *Factory*. If you use the binary format, do the following:
 1. Create a new package with name *Bin[package name]* (for example *BinMyAttributePackage*) containing a class *BinMyAttributePackage_MyAttributeDriver*. The new class inherits *BinMDF_ADriver* class and contains the translation functionality: from transient to persistent and vice versa (see the realization of the standard attributes in the packages *BinMDataStd*, for example). Add package method *AddDrivers*, which adds your class to a driver table.

2. Create a new package (or do it in the current one) with two package methods:
 - Factory, which loads the document storage and retrieval drivers; and
 - AttributeDrivers, which calls the methods AddDrivers for all packages responsible for persistence of the document.
3. Create a plug-in implemented as an executable (see example *BinPlugin*). It calls a macro PLUGIN with the package name where you implemented the method Factory. See *Saving the document and Opening the document from a file* for the description of document save/open mechanisms.

If you decided to use the alternative way (create a new attribute by means of *UAttribute* and a combination of other standard attributes), do the following:

1. Set a *TDataStd_UAttribute* with a unique GUID attached to a label. This attribute defines the semantics of the data type (identifies the data type).
2. Create child labels and allocate all necessary data through standard attributes at the child labels.
3. Define an interface class for access to the data of the child labels.

Choosing the alternative way of implementation of new data types allows to forget about creating persistence classes for your new data type. Standard persistence classes will be used instead. Besides, this way allows separating the data and the methods for access to the data (interfaces). It can be used for rapid development in all cases when requirements to application performance are not very high.

Let's study the implementation of the same data type in both ways by the example of transformation represented by *gp_Trsf* class. The class *gp_Trsf* defines the transformation according to the type (*gp_TrsfForm*) and a set of parameters of the particular type of transformation (two points or a vector for translation, an axis and an angle for rotation, and so on).

1. The first way: creation of a new attribute. The implementation of the transformation by creation of a new attribute is represented in the *Samples*.
2. The second way: creation of a new data type by means of combination of standard attributes. Depending on the type of transformation it may be kept in data framework by different standard attributes. For example, a translation is defined by two points. Therefore the data tree for translation looks like this:
 - Type of transformation (*gp_Translation*) as *TDataStd_Integer*;
 - First point as *TDataStd_RealArray* (three values: X1, Y1 and Z1);
 - Second point as *TDataStd_RealArray* (three values: X2, Y2 and Z2).

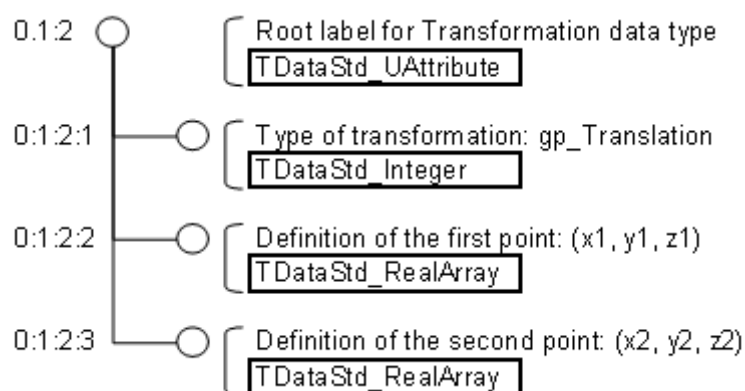


Figure 6: Data tree for translation

If the type of transformation is changed to rotation, the data tree looks like this:

- Type of transformation (*gp_Rotation*) as *TDataStd_Integer*;
- Point of axis of rotation as *TDataStd_RealArray* (three values: X, Y and Z);
- Axis of rotation as *TDataStd_RealArray* (three values: DX, DY and DZ);
- Angle of rotation as *TDataStd_Real*.

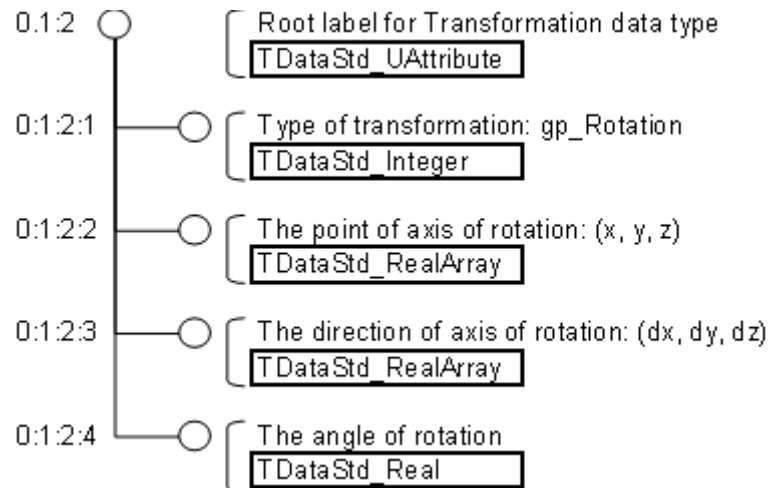


Figure 7: Data tree for rotation

The attribute *TDataStd_UAttribute* with the chosen unique GUID identifies the data type. The interface class initialized by the label of this attribute allows access to the data container (type of transformation and the data of transformation according to the type).

4 Standard Document Services

4.1 Overview

Standard documents offer ready-to-use documents containing a TDF-based data framework. Each document can contain only one framework.

The documents themselves are contained in the instantiation of a class inheriting from *TDocStd_Application*. This application manages the creation, storage and retrieval of documents.

You can implement undo and redo in your document, and refer from the data framework of one document to that of another one. This is done by means of external link attributes, which store the path and the entry of external links.

To sum up, standard documents alone provide access to the data framework. They also allow you to:

- Update external links
- Manage the saving and opening of data
- Manage the undo/redo functionality.

4.2 The Application

As a container for your data framework, you need a document, and your document must be contained in your application. This application will be a class inheriting from *TDocStd_Application*.

4.2.1 Creating an application

To create an application, use the following syntax.

```
Handle(TDocStd_Application) app  
= new MyApplication_Application ();
```

Note that *MyApplication_Application* is a class, which you have to create and which will inherit from *TDocStd_Application*.

4.2.2 Creating a new document

To the application which you declared in the previous example (4.2.1), you must add the document *doc* as an argument of *TDocStd_Application::NewDocument*.

```
Handle(TDocStd_Document) doc;  
app->NewDocument("NewDocumentFormat", doc);
```

4.2.3 Retrieving the application to which the document belongs

To retrieve the application containing your document, you use the syntax below.

```
app = Handle(TDocStd_Application)::DownCast  
(doc->Application());
```

4.3 The Document

The document contains your data framework, and allows you to retrieve this framework, recover its main label, save it in a file, and open or close this file.

4.3.1 Accessing the main label of the framework

To access the main label in the data framework, you use *TDocStd_Document::Main* as in the example below. The main label is the first child of the root label in the data framework, and has the entry 0:1.

```
TDF_Label label = doc->Main();
```

4.3.2 Retrieving the document from a label in its framework

To retrieve the document from a label in its data framework, you use *TDocStd_Document::Get* as in the example below. The argument *label* passed to this method is an instantiation of *TDF_Label*.

```
doc = TDocStd_Document::Get(label);
```

4.3.3 Saving the document

If in your document you use only standard attributes (from the packages *TDF*, *TDataStd*, *TNaming*, *TFunction*, *TPRsStd* and *TDocStd*), you just do the following steps:

- In your application class (which inherits class *TDocStd_Application*) implement two methods:
 - *Formats* (*TColStd_SequenceOfExtendedString& theFormats*), which append to a given sequence *<theFormats>* your document format string, for example, "NewDocumentFormat" – this string is also set in the document creation command
 - *ResourcesName()*, which returns a string with a name of resources file (this file contains a description about the extension of the document, storage/retrieval drivers GUIDs...), for example, "NewFormat"
- Create the resource file (with name, for example, "NewFormat") with the following strings:

```
formatlist:NewDocumentFormat
NewDocumentFormat: New Document Format Version 1.0
NewDocumentFormat.FileExtension: ndf
NewDocumentFormat.StoragePlugin: bd696000-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormat.RetrievalPlugin: bd696001-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormat.Schema: bd696002-5b34-11d1-b5ba-00a0c9064368
NewDocumentFormat.AttributeStoragePlugin: 57b0b826-d931-11d1-b5da-00a0c9064368
NewDocumentFormat.AttributeRetrievalPlugin: 57b0b827-d931-11d1-b5da-00a0c9064368
```

- Create the resource file "Plugin" with GUIDs and corresponding plugin libraries, which looks like this:

Example

```
! Description of available plugins
! *****

b148e300-5740-11d1-a904-080036aaa103.Location: libFWOSPlugin.so
!
! standard document drivers plugin
!
bd696000-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
bd696001-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
!
! standard schema plugin
!
bd696002-5b34-11d1-b5ba-00a0c9064368.Location: libPAppStdPlugin.so
!
! standard attribute drivers plugin
!
57b0b826-d931-11d1-b5da-00a0c9064368.Location: libPAppStdPlugin.so
57b0b827-d931-11d1-b5da-00a0c9064368.Location: libPAppStdPlugin.so
```

In order to set the paths for these files it is necessary to set the environments: *CSF_PluginDefaults* and *CSF_NewFormatDefaults*. For example, set the files in the directory *MyApplicationPath/MyResources*:

```
setenv CSF_PluginDefaults MyApplicationPath/MyResources
setenv CSF_NewFormatDefaults MyApplicationPath/MyResources
```

Once these steps are taken you may run your application, create documents and Save/Open them. These resource files already exist in the OCAF (format "Standard").

If you use your specific attributes from packages, for example, *P*-, *M*- and *TMyAttributePackage* (see Specific attribute creation) you must take some additional steps for the new plugin implementation:

1. Add our *P* package to the standard schema. You can get an already existing (in Open CASCADE Technology sources) schema from *StdSchema* unit and add your package string to the cdl-file: package *PMyAttributePackage*.
2. The next step consists in the implementation of an executable, which will connect our documents to our application and open/save them. Copy the package *PAppStdPlugin* and change its name to *MyTheBestApplicationPlugin*. In the *PLUGIN* macros type the name of your factory, which will be defined at the next step.
3. *Factory* is a method, which returns drivers (standard drivers and our defined drivers from the *M* package) by a GUID. Copy the package to the location, where the standard factory is defined (it is *PAppStd* in the OCAF sources). Change its name to *MyTheBestSchemaLocation*. The *Factory()* method of the *PAppStd* package checks the GUID set as its argument and returns the corresponding table of drivers. Set two new GUIDs for your determined storage and retrieval drivers. Append two *if* declarations inside the *Factory()* method, which should check whether the set GUID coincides with GUIDs defined by the *Factory()* method as far as our storage and retrieval drivers are concerned. If the GUID coincides with one of them, the method should return a table of storage or retrieval drivers respectively.
4. Recompile all and add the strings with GUIDs to the *Plugin* file in accordance with your plugin library GUID.

4.3.4 Opening the document from a file

To open the document from a file where it has been previously saved, you can use *TDocStd_Application::Open* as in the example below. The arguments are the path of the file and the document saved in this file.

```
app->Open("/tmp/example.caf", doc);
```

4.3.5 Cutting, copying and pasting inside a document

To cut, copy and paste inside a document, use the class *TDF_CopyLabel*.

In fact, you must define a *Label*, which contains the temporary value of a cut or copy operation (say, in *Lab_Clipboard*). You must also define two other labels:

- The data container (e.g. *Lab_source*)
- The destination of the copy (e.g. *Lab_target*)

```
Copy = copy (Lab_Source => Lab_Clipboard)
Cut = copy + Lab_Source.ForgetAll() // command clear the contents of LabelSource.
Paste = copy (Lab_Clipboard => Lab_target)
```

So we need a tool to copy all (or a part) of the content of a label and its sub-label, to another place defined by a label.

```
TDF_CopyLabel aCopy;
TDF_IDFilter aFilter (Standard_False);

//Don't copy TDataStd_TreeNode attribute

aFilter.Ignore(TDataStd_TreeNode::GetDefaultTreeID());
aCopy.Load(aSource, aTarget); aCopy.UseFilter(aFilter); aCopy.Perform();

// copy the data structure to clipboard

return aCopy.IsDone(); }
```

The filter is used to forbid copying a specified type of attribute.

You can also have a look at the class *TDF_Closure*, which can be useful to determine the dependencies of the part you want to cut from the document.

4.4 External Links

External links refer from one document to another. They allow you to update the copy of data framework later on.

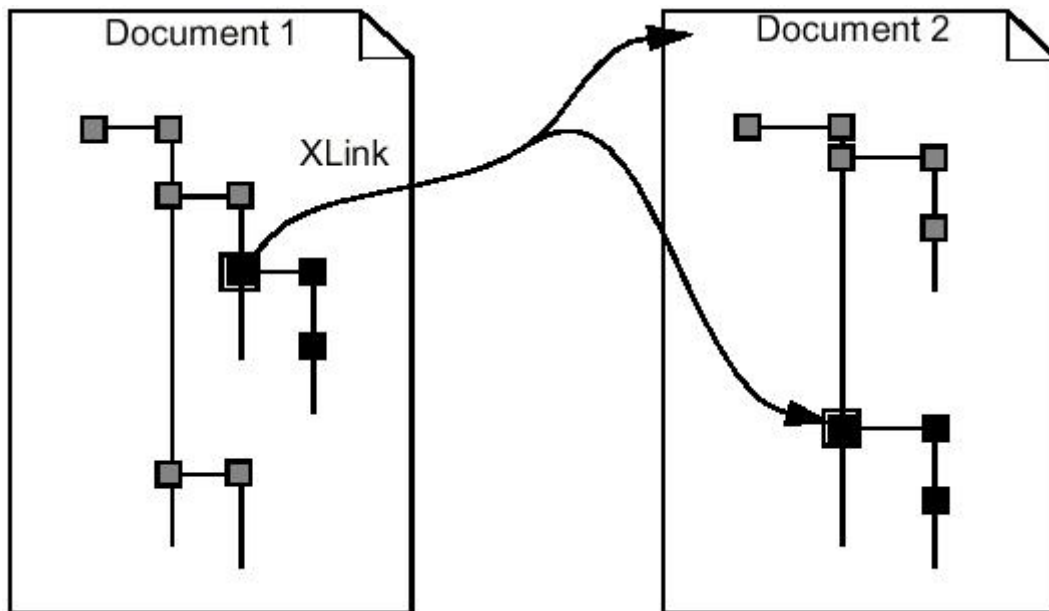


Figure 8: External links between documents

Note that documents can be copied with or without a possibility of updating an external link.

4.4.1 Copying the document

With the possibility of updating it later

To copy a document with a possibility of updating it later, you use *TDocStd_XLinkTool::CopyWithLink*.

```
Handle(TDocStd_Document) doc1;
Handle(TDocStd_Document) doc2;

TDF_Label source = doc1->GetData()->Root();
TDF_Label target = doc2->GetData()->Root();
TDocStd_XLinkTool XLinkTool;

XLinkTool.CopyWithLink(target, source);
```

Now the target document has a copy of the source document. The copy also has a link in order to update the content of the copy if the original changes.

In the example below, something has changed in the source document. As a result, you need to update the copy in the target document. This copy is passed to *TDocStd_XLinkTool::UpdateLink* as the argument *target*.

```
XLinkTool.UpdateLink(target);
```

Without any link between the copy and the original

You can also create a copy of the document with no link between the original and the copy. The syntax to use this option is *TDocStd_XLinkTool::Copy*. The copied document is again represented by the argument *target*, and the original – by *source*.

```
XLinkTool.Copy(target, source);
```

5 OCAF Shape Attributes

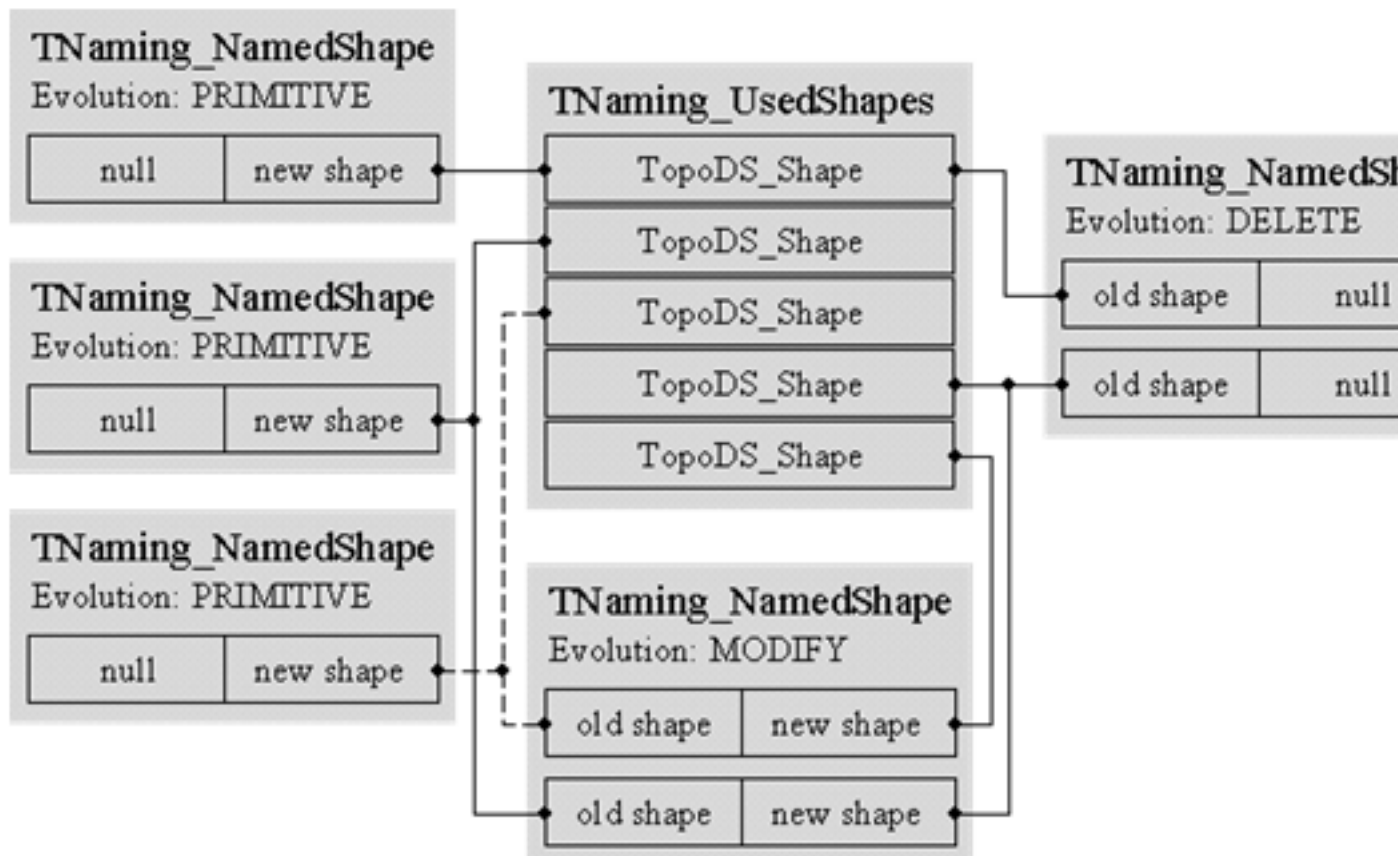
5.1 Overview

A topological attribute can be seen as a hook into the topological structure. It is possible to attach data to define references to it.

OCAF shape attributes are used for topology objects and their evolution access. All topological objects are stored in one *TNaming_UsedShapes* attribute at the root label of the data framework. This attribute contains a map with all topological shapes used in a given document.

The user can add the *TNaming_NamedShape* attribute to other labels. This attribute contains references (hooks) to shapes from the *TNaming_UsedShapes* attribute and an evolution of these shapes. The *TNaming_NamedShape* attribute contains a set of pairs of hooks: to the *Old* shape and to a *New* shape (see the following figure). It allows not only to get the topological shapes by the labels, but also to trace the evolution of the shapes and to correctly update dependent shapes by the changed one.

If a shape is newly created, then the old shape of a corresponding named shape is an empty shape. If a shape is deleted, then the new shape in this named shape is empty.



Shape attributes in data framework.

Different algorithms may dispose sub-shapes of the result shape at the individual labels depending on whether it is necessary to do so:

- If a sub-shape must have some extra attributes (material of each face or color of each edge). In this case a specific sub-shape is placed to a separate label (usually to a sub-label of the result shape label) with all attributes of this sub-shape.

- If the topological naming algorithm is needed, a necessary and sufficient set of sub-shapes is placed to child labels of the result shape label. As usual, for a basic solid and closed shells, all faces of the shape are disposed.

TNaming_NamedShape may contain a few pairs of hooks with the same evolution. In this case the topology shape, which belongs to the named shape is a compound of new shapes.

Consider the following example. Two boxes (solids) are fused into one solid (the result one). Initially each box was placed to the result label as a named shape, which has evolution PRIMITIVE and refers to the corresponding shape of the *TNaming_UsedShapes* map. The box result label has a material attribute and six child labels containing named shapes of Box faces.

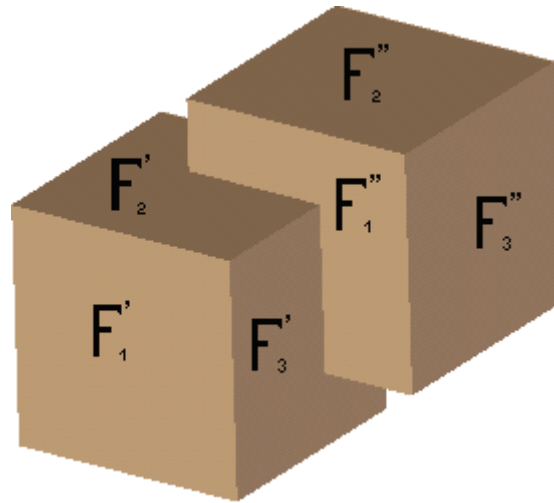
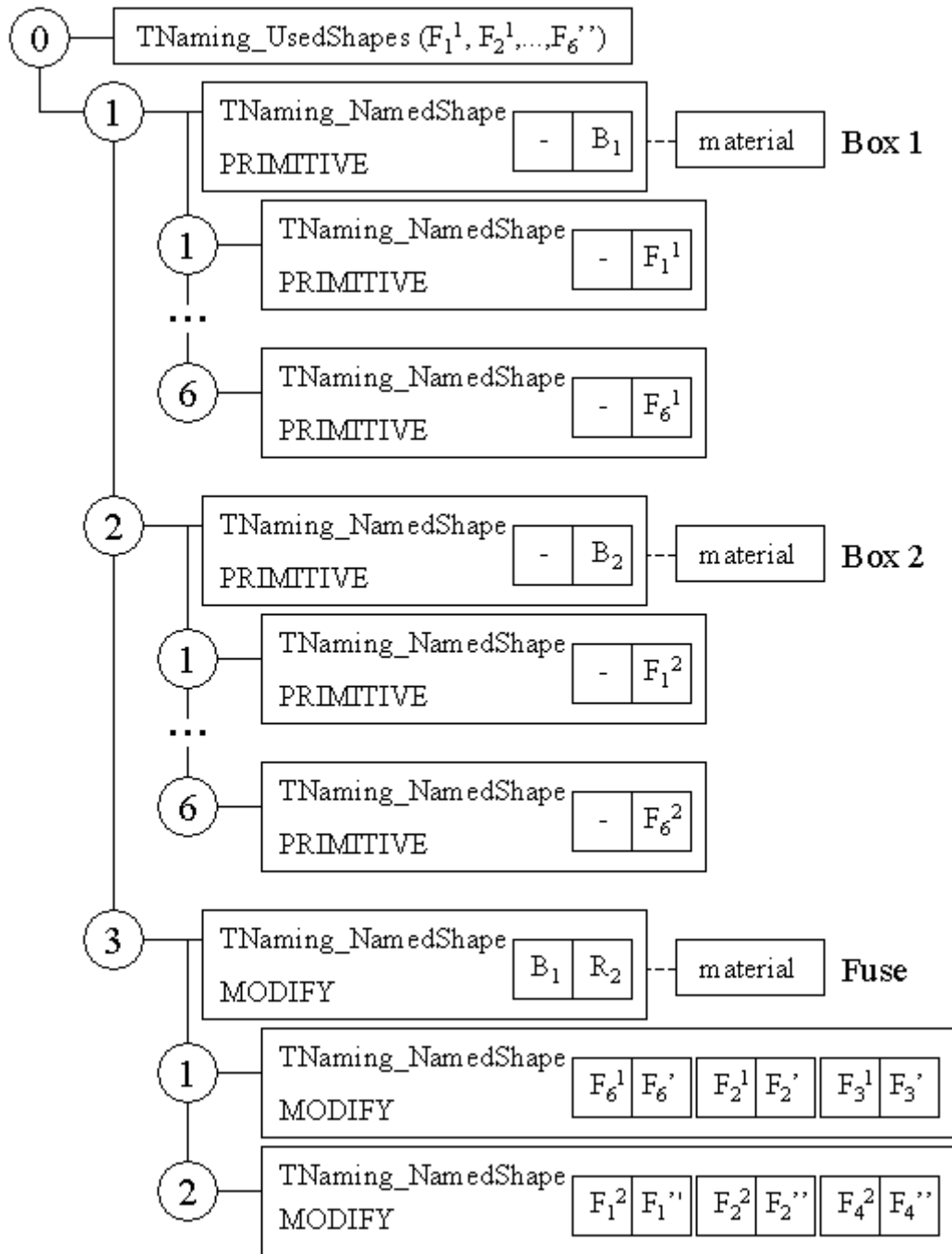


Figure 9: Resulting box

After the fuse operation a modified result is placed to a separate label as a named shape, which refers to the old shape – one of the boxes, as well as to the new shape – the shape resulting from the fuse operation – and has evolution MODIFY (see the following figure).

Named shapes, which contain information about modified faces, belong to the fuse result sub-labels: sub-label with tag 1 – modified faces from box 1, sub-label with tag 2 – modified faces from box 2.



This is necessary and sufficient information for the functionality of the right naming mechanism: any sub-shape of the result can be identified unambiguously by name type and set of labels, which contain named shapes:

- face F_1^1 as a modification of face F_{11}
- face F_1^1 as generation of face F_{12}
- edges as an intersection of two contiguous faces
- vertices as an intersection of three contiguous faces

After any modification of source boxes the application must automatically rebuild the naming entities: recompute the named shapes of the boxes (solids and faces) and fuse the resulting named shapes (solids and faces) that reference to the new named shapes.

5.2 Services provided

5.2.1 Registering shapes and their evolution

When using *TNaming_NamedShape* to create attributes, the following fields of an attribute are filled:

- A list of shapes called the "old" and the "new" shapes A new shape is recomputed as the value of the named shape. The meaning of this pair depends on the type of evolution.
- The type of evolution, which is a term of the *TNaming_Evolution* enumeration used for the selected shapes that are placed to the separate label:
 - PRIMITIVE – newly created topology, with no previous history;
 - GENERATED – as usual, this evolution of a named shape means, that the new shape is created from a low-level old shape (a prism face from an edge, for example);
 - MODIFY – the new shape is a modified old shape;
 - DELETE – the new shape is empty; the named shape with this evolution just indicates that the old shape topology is deleted from the model;
 - SELECTED – a named shape with this evolution has no effect on the history of the topology.

Only pairs of shapes with equal evolution can be stored in one named shape.

5.2.2 Using naming resources

The class *TNaming_Builder* allows you to create a named shape attribute. It has a label of a future attribute as an argument of the constructor. Respective methods are used for the evolution and setting of shape pairs. If for the same *TNaming_Builder* object a lot of pairs of shapes with the same evolution are given, then these pairs would be placed in the resulting named shape. After the creation of a new object of the *TNaming_Builder* class, an empty named shape is created at the given label.

```
// a new empty named shape is created at "label"
TNaming_Builder builder(label);
// set a pair of shapes with evolution GENERATED
builder.Generated(oldshape1,newshape1);
// set another pair of shapes with the same evolution
builder.Generated(oldshape2,newshape2);
// get the result - TNaming_NamedShape attribute
Handle(TNaming_NamedShape) ns = builder.NamedShape();
```

5.2.3 Reading the contents of a named shape attribute

You can use the method *TNaming_NamedShape::Evolution()* to get the evolution of this named shape and the method *TNaming_NamedShape::Get()* to get a compound of new shapes of all pairs of this named shape.

More detailed information about the contents of the named shape or about the modification history of a topology can be obtained with the following:

- *TNaming_Tool* provides a common high-level functionality for access to the named shapes contents:
 - The method *GetShape(Handle(TNaming_NamedShape))* returns a compound of new shapes of the given named shape;
 - The method *CurrentShape(Handle(TNaming_NamedShape))* returns a compound of the shapes, which are latest versions of the shapes from the given named shape;
 - The method *NamedShape(TopoDS_Shape,TDF_Label)* returns a named shape, which contains a given shape as a new shape. A given label is any label from the data framework – it just gives access to it.
- *TNaming_Iterator* gives access to the named shape and hooks pairs.

```
// create an iterator for a named shape
TNaming_Iterator iter(namedshape);
// iterate while some pairs are not iterated
while(iter.More()) {
// get the new shape from the current pair
TopoDS_Shape newshape = iter.NewShape();
// get the old shape from the current pair
TopoDS_Shape oldshape = iter.OldShape();
// do something...

// go to the next pair
iter.Next();
}
```

5.2.4 Selection Mechanism

One of user interfaces for topological naming resources is the *TNaming_Selector* class. You can use this class to:

- Store a selected shape on a label
- Access the named shape
- Update this naming

Selector places a new named shape with evolution SELECTED to the given label. By the given context shape (main shape, which contains a selected sub-shape), its evolution and naming structure the selector creates a "name" of the selected shape – unique description how to find a selected topology.

After any modification of a context shape and updating of the corresponding naming structure, you must call the method *TNaming_Selector::Solve*. If the naming structure is correct, the selector automatically updates the selected shape in the corresponding named shape, else it fails.

5.2.5 Exploring shape evolution

The class *TNaming_Tool* provides a toolkit to read current data contained in the attribute.

If you need to create a topological attribute for existing data, use the method *NamedShape*.

```
class MyPkg_MyClass
{
public: Standard_Boolean SameEdge (const Handle(CafTest_Line)& L1, const Handle(CafTest_Line)& L2);
};

Standard_Boolean CafTest_MyClass::SameEdge (const Handle(CafTest_Line)& L1, const Handle(CafTest_Line)& L2)
{
    Handle(TNaming_NamedShape) NS1 = L1->NamedShape();
    Handle(TNaming_NamedShape) NS2 = L2->NamedShape();
    return BRepTools::Compare(NS1, NS2);
}
```

6 Standard Attributes

6.1 Overview

Standard attributes are ready-to-use attributes, which allow creating and modifying attributes for many basic data types. They are available in the packages *TDataStd*, *TDataXtd* and *TDF*. Each attribute belongs to one of four types:

- Geometric attributes;
- General attributes;
- Relationship attributes;
- Auxiliary attributes.

Geometric attributes

- **Axis** – simply identifies, that the concerned *TNaming_NamedShape* attribute with an axis shape inside belongs to the same label;
- **Constraint** – contains information about a constraint between geometries: used geometry attributes, type, value (if exists), plane (if exists), "is reversed", "is inverted" and "is verified" flags;
- **Geometry** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a specified-type geometry belongs to the same label;
- **Plane** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a plane shape inside belongs to the same label;
- **Point** – simply identifies, that the concerned *TNaming_NamedShape* attribute with a point shape inside belongs to the same label;
- **Shape** – simply identifies, that the concerned *TNaming_NamedShape* attribute belongs to the same label;
- **PatternStd** – identifies one of five available pattern models (linear, circular, rectangular, circular rectangular and mirror);
- **Position** – identifies the position in 3d global space.

General attributes

- **AsciiString** – contains AsciiString value;
- **BooleanArray** – contains an array of Boolean;
- **BooleanList** – contains a list of Boolean;
- **ByteArray** – contains an array of Byte (unsigned char) values;
- **Comment** – contains a string – some comment for a given label (or attribute);
- **Expression** – contains an expression string and a list of used variables attributes;
- **ExtStringArray** – contains an array of *ExtendedString* values;
- **ExtStringList** – contains a list of *ExtendedString* values;
- **Integer** – contains an integer value;
- **IntegerArray** – contains an array of integer values;
- **IntegerList** – contains a list of integer values;

- **IntPackedMap** – contains a packed map of integers;
- **Name** – contains a string – some name of a given label (or attribute);
- **NamedData** – may contain up to 6 of the following named data sets (vocabularies): *DataMapOfStringInteger*, *DataMapOfStringReal*, *DataMapOfStringString*, *DataMapOfStringByte*, *DataMapOfStringHArray1OfInteger* or *DataMapOfStringHArray1OfReal*;
- **NoteBook** – contains a *NoteBook* object attribute;
- **Real** – contains a real value;
- **RealArray** – contains an array of real values;
- **RealList** – contains a list of real values;
- **Relation** – contains a relation string and a list of used variables attributes;
- **Tick** – defines a boolean attribute;
- **Variable** – simply identifies, that a variable belongs to this label; contains the flag *is constraint* and a string of used units ("mm", "m"...);
- **UAttribute** – attribute with a user-defined GUID. As a rule, this attribute is used as a marker, which is independent of attributes at the same label (note, that attributes with the same GUIDs can not belong to the same label).

Relationship attributes

- **Reference** – contains reference to the label of its own data framework;
- **ReferenceArray** – contains an array of references;
- **ReferenceList** – contains a list of references;
- **TreeNode** – this attribute allows to create an internal tree in the data framework; this tree consists of nodes with the specified tree ID; each node contains references to the father, previous brother, next brother, first child nodes and tree ID.

Auxiliary attributes

- **Directory** – high-level tool attribute for sub-labels management;
- **TagSource** – this attribute is used for creation of new children: it stores the tag of the last-created child of the label and gives access to the new child label creation functionality.

All attributes inherit class *TDF_Attribute*, so, each attribute has its own GUID and standard methods for attribute creation, manipulation, getting access to the data framework.

6.2 Services common to all attributes

6.2.1 Accessing GUIDs

To access the GUID of an attribute, you can use two methods:

- Method *GetID* is the static method of a class. It returns the GUID of any attribute, which is an object of a specified class (for example, *TDataStd_Integer* returns the GUID of an integer attribute). Only two classes from the list of standard attributes do not support these methods: *TDataStd_TreeNode* and *TDataStd_Uattribute*, because the GUIDs of these attributes are variable.

- Method *ID* is the method of an object of an attribute class. It returns the GUID of this attribute. Absolutely all attributes have this method: only by this identifier you can discern the type of an attribute.

To find an attribute attached to a specific label, you use the GUID of the attribute type you are looking for. This information can be found using the method *GetID* and the method *Find* for the label as follows:

```
Standard_GUID anID = MyAttributeClass::GetID();  
Standard_Boolean HasAttribute = aLabel.Find(anID,anAttribute);
```

6.2.2 Conventional Interface of Standard Attributes

It is usual to create standard named methods for the attributes:

- Method *Set(label, [value])* is the static method, which allows to add an attribute to a given label. If an attribute is characterized by one value this method may set it.
- Method *Get()* returns the value of an attribute if it is characterized by one value.
- Method *Dump(Standard_OStream)* outputs debug information about a given attribute to a given stream.

To fill a driver table with standard drivers, first initialize the AIS viewer as in the example above, and then pass the return value of the method *InitStandardDrivers* to the driver table returned by the method *Get*. Then attach a *T-Naming_NamedShape* to a label and set the named shape in the presentation attribute using the method *Set*. Then attach the presentation attribute to the named shape attribute, and the *AIS_InteractiveObject*, which the presentation attribute contains, will initialize its drivers for the named shape. This can be seen in the example below.

Example

```
DriverTable::Get() -> InitStandardDrivers();  
// next, attach your named shape to a label  
TPrsStd_AISPresentation::Set(NS);  
// here, attach the AISPresentation to NS.
```

8 Function Services

Function services aggregate data necessary for regeneration of a model. The function mechanism - available in the package *TFunction* - provides links between functions and any execution algorithms, which take their arguments from the data framework, and write their results inside the same framework.

When you edit any application model, you have to regenerate the model by propagating the modifications. Each propagation step calls various algorithms. To make these algorithms independent of your application model, you need to use function services.

Take, for example, the case of a modeling sequence made up of a box with the application of a fillet on one of its edges. If you change the height of the box, the fillet will need to be regenerated as well.

See the white paper *Application Framework Function Mechanism* for more information.

8.1 Finding functions, their owners and roots

The class *TFunction_Function* is an attribute, which stores a link to a function driver in the data framework. In the static table *TFunction_DriverTable* correspondence links between function attributes and drivers are stored.

You can write your function attribute, a driver for such attribute, which updates the function result in accordance to a given map of changed labels, and set your driver with the GUID to the driver table.

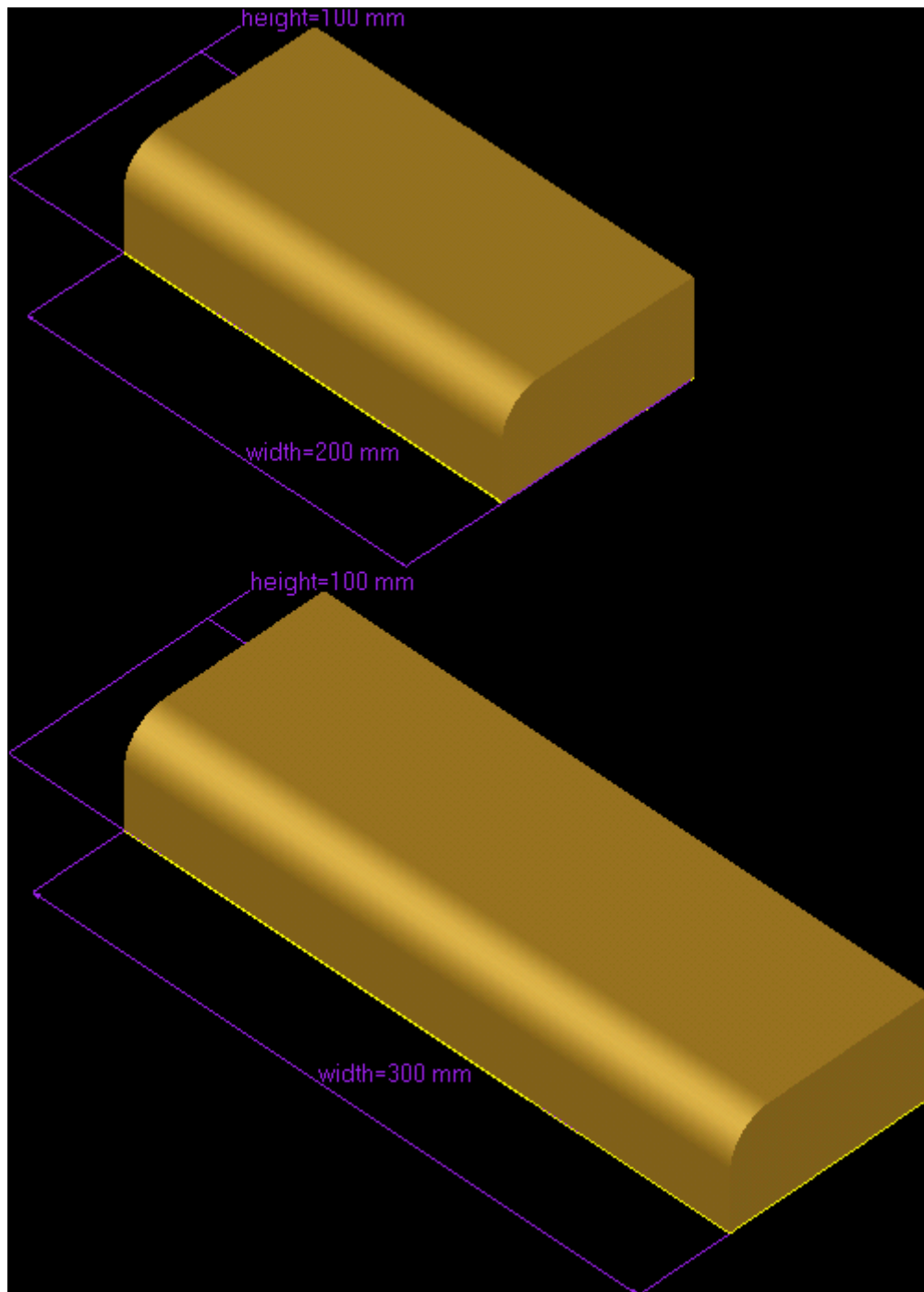
Then the solver algorithm of a data model can find the *Function* attribute on a corresponding label and call the *Execute* driver method to update the result of the function.

8.2 Storing and accessing information about function status

For updating algorithm optimization, each function driver has access to the *TFunction_Logbook* object that is a container for a set of touched, impacted and valid labels. Using this object a driver gets to know which arguments of the function were modified.

8.3 Propagating modifications

An application must implement its functions, function drivers and the common solver for parametric model creation. For example, check the following model:



The procedure of its creation is as follows:

- create a rectangular planar face F with height 100 and width 200
- create prism P using face F as a basis
- create fillet L at the edge of the prism
- change the width of F from 200 to 300:
- the solver for the function of face F starts
- the solver detects that an argument of the face F function has been modified
- the solver calls the driver of the face F function for a regeneration of the face

- the driver rebuilds face F and adds the label of the face *width* argument to the logbook as touched and the label of the function of face F as impacted
- the solver detects the function of P – it depends on the function of F
- the solver calls the driver of the prism P function
- the driver rebuilds prism P and adds the label of this prism to the logbook as impacted
- the solver detects the function of L – it depends on the function of P
- the solver calls the L function driver
- the driver rebuilds fillet L and adds the label of the fillet to the logbook as impacted

9 XML Support

Writing and reading XML files in OCCT is provided by LDOM package, which constitutes an integral part of XML OCAF persistence, which is the optional component provided on top of Open CASCADE Technology.

The Light DOM (LDOM) package contains classes maintaining a data structure whose main principles conform to W3C DOM Level 1 Recommendations. The purpose of these classes as required by XML OCAF persistence schema is to:

- Maintain a tree structure of objects in memory representing the XML document. The root of the structure is an object of the *LDOM_Document* type. This object contains all the data corresponding to a given XML document and contains one object of the *LDOM_Element* type named "document element". The document element contains other *LDOM_Element* objects forming a tree. Other types of nodes: *LDOM_Attr*, *LDOM_Text*, *LDOM_Comment* and *LDOM_CDATASection* - represent the corresponding XML types and serve as branches of the tree of elements.
- Provide class *LDOM_Parser* to read XML files and convert them to *LDOM_Document* objects.
- Provide class *LDOM_XmlWriter* to convert *LDOM_Document* to a character stream in XML format and store it in file.

This package covers the functionality provided by numerous products known as "DOM parsers". Unlike most of them, LDOM was specifically developed to meet the following requirements:

- To minimize the virtual memory allocated by DOM data structures. In average, the amount of memory of LDOM is the same as the XML file size (UTF-8).
- To minimize the time required for parsing and formatting XML, as well as for access to DOM data structures.

Both these requirements are important when XML files are processed by applications if these files are relatively large (occupying megabytes and even hundreds of megabytes). To meet the requirements, some limitations were imposed on the DOM Level 1 specification; these limitations are insignificant in applications like OCAF. Some of these limitations can be overridden in the course of future developments. The main limitations are:

- No Unicode support as well as various other encodings; only ASCII strings are used in DOM/XML. Note: There is a data type *TCollection_ExtendedString* for wide character data. This type is supported by *LDOM_String* as a sequence of numbers.
- Some superfluous methods are deleted: *getPreviousSibling*, *getParentNode*, etc.
- No resolution of XML Entities of any kind
- No support for DTD: the parser just checks for observance of general XML rules and never validates documents.
- Only 5 available types of DOM nodes: *LDOM_Element*, *LDOM_Attr*, *LDOM_Text*, *LDOM_Comment* and *LDOM_CDATASection*.
- No support of Namespaces; prefixed names are used instead of qualified names.
- No support of the interface *DOMException* (no exception when attempting to remove a non-existing node).

LDOM is dependent on Kernel OCCT classes only. Therefore, it can be used outside OCAF persistence in various algorithms where DOM/XML support may be required.

9.1 Document Drivers

The drivers for document storage and retrieval manage conversion between a transient OCAF Document in memory and its persistent reflection in a container (disk, memory, network). For XML Persistence, they are defined in the package *XmlDrivers*.

The main methods (entry points) of these drivers are:

- *Write()* - for a storage driver;
- *Read()* - for a retrieval driver.

The most common case (which is implemented in XML Persistence) is writing/reading document to/from a regular OS file. Such conversion is performed in two steps:

First it is necessary to convert the transient document into another form (called persistent), suitable for writing into a file, and vice versa. In XML Persistence *LDOM_Document* is used as the persistent form of an OCAF Document and the *DOM_Nodes* are the persistent objects. An OCAF Document is a tree of labels with attributes. Its transformation into a persistent form can be functionally divided into two parts:

- Conversion of the labels structure, which is performed by the method *XmlMDF::FromTo()*
- Conversion of the attributes and their underlying objects, which is performed by the corresponding attribute drivers (one driver per attribute type).

The driver for each attribute is selected from a table of drivers, either by attribute type (on storage) or by the name of the corresponding *DOM_Element* (on retrieval). The table of drivers is created by methods *XmlDrivers_DocumentStorageDriver::AttributeDrivers()* and *XmlDrivers_DocumentRetrievalDriver::AttributeDrivers()*.

Then the persistent document is written into a file (or read from a file). In standard persistence *Storage* and *FSD* packages contain classes for writing/reading the persistent document into a file. In XML persistence *LDOMParser* and *LDOM_XmlWriter* are used instead.

Usually, the library containing document storage and retrieval drivers is loaded at run time by a plugin mechanism. To support this in XML Persistence, there is a plugin *XmlPlugin* and a *Factory()* method in the *XmlDrivers* package. This method compares passed GUIDs with known GUIDs and returns the corresponding driver or generates an exception if the GUID is unknown.

The application defines which GUID is needed for document storage or retrieval and in which library it should be found. This depends on document format and application resources. Resources for XML Persistence and also for standard persistence are found in the *StdResource* unit. They are written for the *XmlOcaf* document format.

9.2 Attribute Drivers

There is one attribute driver for XML persistence for each transient attribute from a set of standard OCAF attributes, with the exception of attribute types, which are never stored (pure transient). Standard OCAF attributes are collected in six packages, and their drivers also follow this distribution. Driver for attribute *T*_** is called *XmlM*_**. Conversion between transient and persistent form of attribute is performed by two methods *Paste()* of attribute driver.

XmlMDF_ADriver is the root class for all attribute drivers.

At the beginning of storage/retrieval process, one instance of each attribute driver is created and appended to driver table implemented as *XmlMDF_ADriverTable*. During OCAF Data storage, attribute drivers are retrieved from the driver table by the type of attribute. In the retrieval step, a data map is created linking names of *DOM_Elements* and attribute drivers, and then attribute drivers are sought in this map by *DOM_Element* qualified tag names.

Every transient attribute is saved as a *DOM_Element* (root element of OCAF attribute) with attributes and possibly sub-nodes. The name of the root element can be defined in the attribute driver as a string passed to the base class constructor. The default is the attribute type name. Similarly, namespace prefixes for each attribute can be set. There is no default value, but it is possible to pass NULL or an empty string to store attributes without namespace prefixes.

The basic class *XmlMDF_ADriver* supports errors reporting via the method *WriteMessage(const TCollection_ExtendedString&)*. It sends a message string to its message driver which is initialized in the constructor with a *Handle(CDM_MessageDriver)* passed from the application by Document Storage/Retrieval Driver.

9.3 XML Document Structure

Every XML Document has one root element, which may have attributes and contain other nodes. In OCAF XML Documents the root element is named "document" and has attribute "format" with the name of the OCAF Schema

used to generate the file. The standard XML format is "XmlOcaf". The following elements are sub-elements of `<document>` and should be unique entries as its sub-elements, in a specific order. The order is:

- **Element info** - contains strings identifying the format version and other parameters of the OCAF XML document. Normally, data under the element is used by persistence algorithms to correctly retrieve and initialize an OCAF document. The data also includes a copyright string.
- **Element comments** - consists of an unlimited number of `<comment>` sub-elements containing necessary comment strings.
- **Element label** is the root label of the document data structure, with the XML attribute "tag" equal to 0. It contains all the OCAF data (labels, attributes) as tree of XML elements. Every sub-label is identified by a tag (positive integer) defining a unique key for all sub-labels of a label. Every label can contain any number of elements representing OCAF attributes (see OCAF Attributes Representation below).
- **Element shapes** - contains geometrical and topological entities in BRep format. These entities being referenced by OCAF attributes written under the element `<label>`. This element is empty if there are no shapes in the document. It is only output if attribute driver *XmlMNaming_NamedShapeDriver* has been added to drivers table by the *DocumentStorageDriver*.

OCAF Attributes Representation

In XML documents, OCAF attributes are elements whose name identifies the OCAF attribute type. These elements may have a simple (string or number) or complex (sub-elements) structure, depending on the architecture of OCAF attribute. Every XML type for OCAF attribute possesses a unique positive integer "id" XML attribute identifying the OCAF attribute throughout the document. To ensure "id" uniqueness, the attribute name "id" is reserved and is only used to indicate and identify elements which may be referenced from other parts of the OCAF XML document. For every standard OCAF attribute, its XML name matches the name of a C++ class in Transient data model. Generally, the XML name of OCAF attribute can be specified in the corresponding attribute driver. XML types for OCAF attributes are declared with XML W3C Schema in a few XSD files where OCAF attributes are grouped by the package where they are defined.

Example of resulting XML file

The following example is a sample text from an XML file obtained by storing an OCAF document with two labels (0: and 0:2) and two attributes - *TDataStd_Name* (on label 0:) and *TNaming_NamedShape* (on label 0:2). The `<shapes>` section contents are replaced by an ellipsis.

```
<?xml version="1.0" encoding="UTF-8"?>
<document format="XmlOcaf" xmlns="http://www.opencascade.org/OCAF/XML" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opencascade.org/OCAF/XML http://www.opencascade.org/OCAF/XML/XmlOcaf.xsd">

  <info date="2001-10-04" schemav="0" objnb="3">
    <iitem>Copyright: Open Cascade, 2001</iitem>
    <iitem>STORAGE_VERSION: PCDM_ReadWriter_1</iitem>
    <iitem>REFERENCE_COUNTER: 0</iitem>
    <iitem>MODIFICATION_COUNTER: 1</iitem>
  </info>
  <comments/>
  <label tag="0">
    <TDataStd_Name id="1">Document_1</TDataStd_Name>
    <label tag="2">
      <TNaming_NamedShape id="2" evolution="primitive">
        <olds/>
        <news>
          <shape tshape="+34" index="1"/>
        </news>
      </TNaming_NamedShape>
    </label>
  </label>
  \<shapes\>
  ...
</shapes>
</document>
```

9.4 XML Schema

The XML Schema defines the class of a document.

The full structure of OCAF XML documents is described as a set of XML W3C Schema files with definitions of all XML element types. The definitions provided cannot be overridden. If any application defines new persistence schemas, it can use all the definitions from the present XSD files but if it creates new or redefines existing types, the definition must be done under other namespace(s).

There are other ways to declare XML data, different from W3C Schema, and it should be possible to use them to the extent of their capabilities of expressing the particular structure and constraints of our XML data model. However, it must be noted that the W3C Schema is the primary format for declarations and as such, it is the format supported for future improvements of Open CASCADE Technology, including the development of specific applications using OCAF XML persistence.

The Schema files (XSD) are intended for two purposes:

- documenting the data format of files generated by OCAF;
- validation of documents when they are used by external (non-OCAF) applications, e.g., to generate reports.

The Schema definitions are not used by OCAF XML Persistence algorithms when saving and restoring XML documents. There are internal checks to ensure validity when processing every type of data.

Management of Namespaces

Both the XML format and the XML OCAF persistence code are extensible in the sense that every new development can reuse everything that has been created in previous projects. For the XML format, this extensibility is supported by assigning names of XML objects (elements) to different XML Namespaces. Hence, XML elements defined in different projects (in different persistence libraries) can easily be combined into the same XML documents. An example is the XCAF XML persistence built as an extension to the Standard OCAF XML persistence [*File XmlXcaf.xsd*]. For the correct management of Namespaces it is necessary to:

- Define *targetNamespace* in the new XSD file describing the format.
- Declare (in XSD files) all elements and types in the *targetNamespace* to appear without a namespace prefix; all other elements and types use the appropriate prefix (such as "ocaf:").
- Add (in the new *DocumentStorageDriver*) the *targetNamespace* accompanied with its prefix, using method *XmlDrivers_DocumentStorageDriver::AddNamespace*. The same is done for all namespaces objects which are used by the new persistence, with the exception of the "ocaf" namespace.
- Pass (in every OCAF attribute driver) the namespace prefix of the *targetNamespace* to the constructor of *XmlMDF_ADriver*.

10 GLOSSARY

- **Application** - a document container holding all documents containing all application data.
- **Application data** - the data produced by an application, as opposed to data referring to it.
- **Associativity of data** - the ability to propagate modifications made to one document to other documents, which refer to such document. Modification propagation is:
 - unidirectional, that is, from the referenced to the referencing document(s), or
 - bi-directional, from the referencing to the referenced document and vice-versa.
- **Attribute** - a container for application data. An attribute is attached to a label in the hierarchy of the data framework.
- **Child** - a label created from another label, which by definition, is the father label.
- **Compound document** - a set of interdependent documents, linked to each other by means of external references. These references provide the associativity of data.
- **Data framework** - a tree-like data structure which in OCAF, is a tree of labels with data attached to them in the form of attributes. This tree of labels is accessible through the services of the *TDocStd_Document* class.
- **Document** - a container for a data framework which grants access to the data, and is, in its turn, contained by an application. A document also allows you to:
 - Manage modifications, providing Undo and Redo functions
 - Manage command transactions
 - Update external links
 - Manage save and restore options
 - Store the names of software extensions.
- **Driver** - an abstract class, which defines the communications protocol with a system.
- **Entry** - an ASCII character string containing the tag list of a label. For example:

0:3:24:7:2:7

- **External links** - references from one data structure to another data structure in another document. To store these references properly, a label must also contain an external link attribute.
- **Father** - a label, from which other labels have been created. The other labels are, by definition, the children of this label.
- **Framework** - a group of co-operating classes which enable a design to be re-used for a given category of problem. The framework guides the architecture of the application by breaking it up into abstract classes, each of which has different responsibilities and collaborates in a predefined way. Application developer creates a specialized framework by:
 - defining new classes which inherit from these abstract classes
 - composing framework class instances
 - implementing the services required by the framework.

In C++, the application behavior is implemented in virtual functions redefined in these derived classes. This is known as overriding.

- **GUID** - Global Universal ID. A string of 37 characters intended to uniquely identify an object. For example:

2a96b602-ec8b-11d0-bee7-080009dc3333

- **Label** - a point in the data framework, which allows data to be attached to it by means of attributes. It has a name in the form of an entry, which identifies its place in the data framework.
- **Modified label** - containing attributes whose data has been modified.
- **Reference key** - an invariant reference, which may refer to any type of data used in an application. In its transient form, it is a label in the data framework, and the data is attached to it in the form of attributes. In its persistent form, it is an entry of the label. It allows an application to recover any entity in the current session or in a previous session.
- **Resource file** - a file containing a list of each document's schema name and the storage and retrieval plug-ins for that document.
- **Root** - the starting point of the data framework. This point is the top label in the framework. It is represented by the [0] entry and is created at the same time with the document you are working on.
- **Scope** - the set of all the attributes and labels which depend on a given label.
- **Tag list** - a list of integers, which identify the place of a label in the data framework. This list is displayed in an entry.
- **Topological naming** - systematic referencing of topological entities so that these entities can still be identified after the models they belong to have gone through several steps in modeling. In other words, topological naming allows you to track entities through the steps in the modeling process. This referencing is needed when a model is edited and regenerated, and can be seen as a mapping of labels and name attributes of the entities in the old version of a model to those of the corresponding entities in its new version. Note that if the topology of a model changes during the modeling, this mapping may not fully coincide. A Boolean operation, for example, may split edges.
- **Topological tracking** - following a topological entity in a model through the steps taken to edit and regenerate that model.
- **Valid label** - in a data framework, this is a label, which is already recomputed in the scope of regeneration sequence and includes the label containing a feature which is to be recalculated. Consider the case of a box to which you first add a fillet, then a protrusion feature. For recalculation purposes, only valid labels of each construction stage are used. In recalculating a fillet, they are only those of the box and the fillet, not the protrusion feature which was added afterwards.

11 Samples

11.1 Implementation of Attribute Transformation in a CDL file

```

class Transformation from MyPackage inherits Attribute from TDF

  ---Purpose: This attribute implements a transformation data container.

uses

  Attribute      from TDF,
  Label          from TDF,
  GUID           from Standard,
  RelocationTable from TDF,
  Pnt            from gp,
  Ax1            from gp,
  Ax2            from gp,
  Ax3            from gp,
  Vec            from gp,
  Trsf           from gp,
  TrsfForm       from gp

is

  ---Category: Static methods
  --
  =====

  GetID (myclass)
  ---C++: return const &
  ---Purpose: The method returns a unique GUID of this attribute.
  --           By means of this GUID this attribute may be identified
  --           among other attributes attached to the same label.
  returns GUID from Standard;

  Set (myclass; theLabel : Label from TDF)
  ---Purpose: Finds or creates the attribute attached to <theLabel>.
  --           The found or created attribute is returned.
  returns Transformation from MyPackage;

  ---Category: Methods for access to the attribute data
  --
  =====

  Get (me)
  ---Purpose: The method returns the transformation.
  returns Trsf from gp;

  ---Category: Methods for setting the data of transformation
  --
  =====

  SetRotation (me : mutable;
               theAxis : Ax1 from gp;
               theAngle : Real from Standard);
  ---Purpose: The method defines a rotation type of transformation.

  SetTranslation (me : mutable;
                 theVector : Vec from gp);
  ---Purpose: The method defines a translation type of transformation.

  SetMirror (me : mutable;
             thePoint : Pnt from gp);
  ---Purpose: The method defines a point mirror type of transformation
  --           (point symmetry).

  SetMirror (me : mutable;
             theAxis : Ax1 from gp);
  ---Purpose: The method defines an axis mirror type of transformation
  --           (axial symmetry).

  SetMirror (me : mutable;
             thePlane : Ax2 from gp);
  ---Purpose: The method defines a point mirror type of transformation
  --           (planar symmetry).

  SetScale (me : mutable;
            thePoint : Pnt from gp;
            theScale : Real from Standard);
  ---Purpose: The method defines a scale type of transformation.

  SetTransformation (me : mutable;
                    theCoordinateSystem1 : Ax3 from gp;
                    theCoordinateSystem2 : Ax3 from gp);
  ---Purpose: The method defines a complex type of transformation
  --           from one co-ordinate system to another.

```

```

---Category: Overridden methods from TDF_Attribute
--
=====

ID (me)
---C++: return const &
---Purpose: The method returns a unique GUID of the attribute.
--           By means of this GUID this attribute may be identified
--           among other attributes attached to the same label.
returns GUID from Standard;

Restore (me: mutable;
        theAttribute : Attribute from TDF);
---Purpose: The method is called on Undo / Redo.
--           It copies the content of <theAttribute>
--           into this attribute (copies the fields).

NewEmpty (me)
---Purpose: It creates a new instance of this attribute.
--           It is called on Copy / Paste, Undo / Redo.
returns mutable Attribute from TDF;

Paste (me;
       theAttribute : mutable Attribute from TDF;
       theRelocationTable : mutable RelocationTable from TDF);
---Purpose: The method is called on Copy / Paste.
--           It copies the content of this attribute into
--           <theAttribute> (copies the fields).

Dump (me; anOS : in out OStream from Standard)
---C++: return;
---Purpose: Prints the content of this attribute into the stream.
returns OStream from Standard is redefined;

---Category: Constructor
--
=====

Create
---Purpose: The C++ constructor of this attribute class.
--           Usually it is never called outside this class.
returns mutable Transformation from MyPackage;

fields

-- Type of transformation
myType : TrsfForm from gp;

-- Axes (Ax1, Ax2, Ax3)
myAx1 : Ax1 from gp;
myAx2 : Ax2 from gp;
myFirstAx3 : Ax3 from gp;
mySecondAx3 : Ax3 from gp;

-- Scalar values
myAngle : Real from Standard;
myScale : Real from Standard;

-- Points
myFirstPoint : Pnt from gp;
mySecondPoint : Pnt from gp;

end Transformation;

```

11.2 Implementation of Attribute Transformation in a CPP file

```

#include MyPackage_Transformation.ixx;

//=====
//function : GetID
//purpose   : The method returns a unique GUID of this attribute.
//           By means of this GUID this attribute may be identified
//           among other attributes attached to the same label.
//=====
const Standard_GUID& MyPackage_Transformation::GetID()
{
    static Standard_GUID ID("4443368E-C808-4468-984D-B26906BA8573");
    return ID;
}

//=====

```

```

//function : Set
//purpose : Finds or creates the attribute attached to <theLabel>.
// The found or created attribute is returned.
//=====
Handle(MyPackage_Transformation) MyPackage_Transformation::Set(const TDF_Label& theLabel)
{
    Handle(MyPackage_Transformation) T;
    if (!theLabel.FindAttribute(MyPackage_Transformation::GetID(), T))
    {
        T = new MyPackage_Transformation();
        theLabel.AddAttribute(T);
    }
    return T;
}

//=====
//function : Get
//purpose : The method returns the transformation.
//=====
gp_Trsf MyPackage_Transformation::Get() const
{
    gp_Trsf transformation;
    switch (myType)
    {
        case gp_Identity:
        {
            break;
        }
        case gp_Rotation:
        {
            transformation.SetRotation(myAx1, myAngle);
            break;
        }
        case gp_Translation:
        {
            transformation.SetTranslation(myFirstPoint, mySecondPoint);
            break;
        }
        case gp_PntMirror:
        {
            transformation.SetMirror(myFirstPoint);
            break;
        }
        case gp_Ax1Mirror:
        {
            transformation.SetMirror(myAx1);
            break;
        }
        case gp_Ax2Mirror:
        {
            transformation.SetMirror(myAx2);
            break;
        }
        case gp_Scale:
        {
            transformation.SetScale(myFirstPoint, myScale);
            break;
        }
        case gp_CompoundTrsf:
        {
            transformation.SetTransformation(myFirstAx3, mySecondAx3);
            break;
        }
        case gp_Other:
        {
            break;
        }
    }
    return transformation;
}

//=====
//function : SetRotation
//purpose : The method defines a rotation type of transformation.
//=====
void MyPackage_Transformation::SetRotation(const gp_Ax1& theAxis, const Standard_Real theAngle)
{
    Backup();
    myType = gp_Rotation;
    myAx1 = theAxis;
    myAngle = theAngle;
}

//=====
//function : SetTranslation
//purpose : The method defines a translation type of transformation.
//=====

```

```

void MyPackage_Transformation::SetTranslation(const gp_Vec& theVector)
{
    Backup();
    myType = gp_Translation;
    myFirstPoint.SetCoord(0, 0, 0);
    mySecondPoint.SetCoord(theVector.X(), theVector.Y(), theVector.Z());
}

//=====
//function : SetMirror
//purpose : The method defines a point mirror type of transformation
//          (point symmetry).
//=====
void MyPackage_Transformation::SetMirror(const gp_Pnt& thePoint)
{
    Backup();
    myType = gp_PntMirror;
    myFirstPoint = thePoint;
}

//=====
//function : SetMirror
//purpose : The method defines an axis mirror type of transformation
//          (axial symmetry).
//=====
void MyPackage_Transformation::SetMirror(const gp_Ax1& theAxis)
{
    Backup();
    myType = gp_Ax1Mirror;
    myAx1 = theAxis;
}

//=====
//function : SetMirror
//purpose : The method defines a point mirror type of transformation
//          (planar symmetry).
//=====
void MyPackage_Transformation::SetMirror(const gp_Ax2& thePlane)
{
    Backup();
    myType = gp_Ax2Mirror;
    myAx2 = thePlane;
}

//=====
//function : SetScale
//purpose : The method defines a scale type of transformation.
//=====
void MyPackage_Transformation::SetScale(const gp_Pnt& thePoint, const Standard_Real theScale)
{
    Backup();
    myType = gp_Scale;
    myFirstPoint = thePoint;
    myScale = theScale;
}

//=====
//function : SetTransformation
//purpose : The method defines a complex type of transformation
//          from one co-ordinate system to another.
//=====
void MyPackage_Transformation::SetTransformation(const gp_Ax3& theCoordinateSystem1,
                                                const gp_Ax3& theCoordinateSystem2)
{
    Backup();
    myFirstAx3 = theCoordinateSystem1;
    mySecondAx3 = theCoordinateSystem2;
}

//=====
//function : ID
//purpose : The method returns a unique GUID of the attribute.
//          By means of this GUID this attribute may be identified
//          among other attributes attached to the same label.
//=====
const Standard_GUID& MyPackage_Transformation::ID() const
{
    return GetID();
}

//=====
//function : Restore
//purpose : The method is called on Undo / Redo.
//          It copies the content of <theAttribute>
//          into this attribute (copies the fields).
//=====

```

```

void MyPackage_Transformation::Restore(const Handle(TDF_Attribute)& theAttribute)
{
    Handle(MyPackage_Transformation) theTransformation = Handle(MyPackage_Transformation)::DownCast (
        theAttribute);
    myType = theTransformation->myType;
    myAx1 = theTransformation->myAx1;
    myAx2 = theTransformation->myAx2;
    myFirstAx3 = theTransformation->myFirstAx3;
    mySecondAx3 = theTransformation->mySecondAx3;
    myAngle = theTransformation->myAngle;
    myScale = theTransformation->myScale;
    myFirstPoint = theTransformation->myFirstPoint;
    mySecondPoint = theTransformation->mySecondPoint;
}

//=====
//function : NewEmpty
//purpose : It creates a new instance of this attribute.
//          It is called on Copy / Paste, Undo / Redo.
//=====
Handle(TDF_Attribute) MyPackage_Transformation::NewEmpty() const
{
    return new MyPackage_Transformation();
}

//=====
//function : Paste
//purpose : The method is called on Copy / Paste.
//          It copies the content of this attribute into
//          <theAttribute> (copies the fields).
//=====
void MyPackage_Transformation::Paste(const Handle(TDF_Attribute)& theAttribute,
                                     const Handle(TDF_RelocationTable)& ) const
{
    Handle(MyPackage_Transformation) theTransformation = Handle(MyPackage_Transformation)::DownCast (
        theAttribute);
    theTransformation->myType = myType;
    theTransformation->myAx1 = myAx1;
    theTransformation->myAx2 = myAx2;
    theTransformation->myFirstAx3 = myFirstAx3;
    theTransformation->mySecondAx3 = mySecondAx3;
    theTransformation->myAngle = myAngle;
    theTransformation->myScale = myScale;
    theTransformation->myFirstPoint = myFirstPoint;
    theTransformation->mySecondPoint = mySecondPoint;
}

//=====
//function : Dump
//purpose : Prints the content of this attribute into the stream.
//=====
Standard_OStream& MyPackage_Transformation::Dump(Standard_OStream& anOS) const
{
    anOS = "Transformation: ";
    switch (myType)
    {
        case gp_Identity:
        {
            anOS = "gp_Identity";
            break;
        }
        case gp_Rotation:
        {
            anOS = "gp_Rotation";
            break;
        }
        case gp_Translation:
        {
            anOS = "gp_Translation";
            break;
        }
        case gp_PntMirror:
        {
            anOS = "gp_PntMirror";
            break;
        }
        case gp_Ax1Mirror:
        {
            anOS = "gp_Ax1Mirror";
            break;
        }
        case gp_Ax2Mirror:
        {
            anOS = "gp_Ax2Mirror";
            break;
        }
        case gp_Scale:
    }
}

```

```

    {
        anOS = "gp_Scale";
        break;
    }
    case gp_CompoundTrsf:
    {
        anOS = "gp_CompoundTrsf";
        break;
    }
    case gp_Other:
    {
        anOS = "gp_Other";
        break;
    }
    }
    return anOS;
}

//=====
//function : MyPackage_Transformation
//purpose  : A constructor.
//=====
MyPackage_Transformation::MyPackage_Transformation():myType(gp_Identity) {
}

```

11.3 Implementation of typical actions with standard OCAF attributes.

There are four sample files provided in the directory 'OpenCasCade/ros/samples/ocafsamples'. They present typical actions with OCAF services (mainly for newcomers). The method *Sample()* of each file is not dedicated for execution 'as is', it is rather a set of logical actions using some OCAF services.

TDataStd_Sample.cxx

This sample contains templates for typical actions with the following standard OCAF attributes:

- Starting with data framework;
- TDataStd_Integer attribute management;
- TDataStd_RealArray attribute management;
- TDataStd_Comment attribute management;
- TDataStd_Name attribute management;
- TDataStd_UAttribute attribute management;
- TDF_Reference attribute management;
- TDataXtd_Point attribute management;
- TDataXtd_Plane attribute management;
- TDataXtd_Axis attribute management;
- TDataXtd_Geometry attribute management;
- TDataXtd_Constraint attribute management;
- TDataStd_Directory attribute management;
- TDataStd_TreeNode attribute management.

TDocStd_Sample.cxx

This sample contains template for the following typical actions:

- creating application;
- creating the new document (document contains a framework);
- retrieving the document from a label of its framework;
- filling a document with data;
- saving a document in the file;
- closing a document;
- opening the document stored in the file;
- copying content of a document to another document with possibility to update the copy in the future.

TPrsStd_Sample.cxx

This sample contains template for the following typical actions:

- starting with data framework;
- setting the TPrsStd_AISViewer in the framework;
- initialization of aViewer;
- finding TPrsStd_AISViewer attribute in the DataFramework;
- getting AIS_InteractiveContext from TPrsStd_AISViewer;
- adding driver to the map of drivers;
- getting driver from the map of drivers;
- setting TNaming_NamedShape to <ShapeLabel>;
- setting the new TPrsStd_AISPresentation to <ShapeLabel>;
- displaying;
- erasing;
- updating and displaying presentation of the attribute to be displayed;
- setting a color to the displayed attribute;
- getting transparency of the displayed attribute;
- modify attribute;
- updating presentation of the attribute in viewer.

TNaming_Sample.cxx

This sample contains template for typical actions with OCAF Topological Naming services. The following scenario is used:

- data framework initialization;
- creating Box1 and pushing it as PRIMITIVE in DF;
- creating Box2 and pushing it as PRIMITIVE in DF;
- moving Box2 (applying a transformation);
- pushing the selected edges of the top face of Box1 in DF;
- creating a Fillet (using the selected edges) and pushing the result as a modification of Box1;
- creating a Cut (Box1, Box2) as a modification of Box1 and push it in DF;
- recovering the result from DF.