Active Data Framework

Developer's Guide

Generated by Doxygen 1.8.9.1

Contents

Chapter 1

Preface

Note

> From time to time in this paper we use AD abbreviation. Keep in mind that AD is Active Data, not Algorithmic Differentiation (the another commonly used term in CAD world).

Active Data framework is a tool set which allows you to organize the data of your application in a hierarchical way with automatic support of the following commonly used functionality:

- *Document architecture:* the data hierarchy is called a *document* in order to emphasize that it can be saved to disk and opened from file. The main storage format is the compact binary, however, this can easily reconfigured to text formats like XML.
- *Undo/Redo:* all changes introduced to the working document can be rolled back (and forward) thanks to the conventional transactional approach. This conception is very close to what we know from relational databases.
- *Copy/Paste:* sometimes the interrelations between your data objects are more complicated than simple parent-child hierarchy. For example, one object can have a reference to another one and hold a back-reference from some third object. These relations are not easy to track once your object is copied or moved in the project hierarchy. Active Data comes with a flexible mechanism which gives you transparent rules of such relocations.
- *Dependency graphs:* this is the core principle giving our framework its name. We use to divide all possible data models by two classes: active and passive. A passive model is nothing but a speechless container of data, which does not provide any reaction to changes in the stored objects. This is the simplest philosophy of data organization which is sometimes not very convenient. Another approach consists in letting your data model react to the changes introduced by user. This can be achieved by a concept of asynchronous events, callback functions or whatever else. The main idea is that your data model stops being a silent bank of data: it can initiate some algorithmic workflows within your software. We call such models active. The activeness principle is realized in AD framework by introducing a notion of execution graph which binds your objects into a dependency network. Each node in this graph can be seen as an abstract calculation launched after its predecessors and initiating launch of its successors.

Choosing the way of data model organization determines to a very high extent the overall architecture of your product. In engineering software (especially if it deals with geometric modelling) all the functionalities listed above are kind of standard. Different software components available on the market address these common needs in different ways. E.g. in some libraries the central object is a CAD shape which may contain user-specific attributes (colors, materials etc), can save and restore itself from file and provides an embedded undo/redo mechanism. Active Data model is an Open CASCADE Technology based product which does not concentrate on CAD geometry and gives a common solution to the problem of data organization.

Active Data framework is based on OCAF (Open CASCADE Application Framework). OCAF is not specific to geometry and allows to describe any kind of data in terms of the so called *Labels* and *Attributes*. Before you continue reading this document, we recommend you to shortly familiarize yourself with the fundamental OCAF principles. Doing so, you will understand better the pros and cons of using a super-layer under OCAF. The main ideas behind AD are as follows:

- OCAF is a powerful tool which can express almost any kind of engineering data. This generality comes at a price: you have to assemble your data model from series of low-level "bricks". And without a good experience this job stops being trivial even in cases of moderate complexity. We wanted to bring more object oriented approach to OCAF-driven design process: Active Data allows you to forget about the atomic operations of OCAF.
- It is always a problem how to maintain all interrelations between your objects as long as your data model evolves. E.g. we want to be sure that no memory leaks occur on deletion of a data object. Similarly, if one object is moved from one branch of the project hierarchy to another, we need to ensure that all possible references are consistently actualized.
- We want to have a unified storage format of our data. OCAF allows you to organize your objects in any order, so it is easy to end up with a complete mess in the internal data organization. Active Data uses a unified convention which presumes that all objects of the same type are stored internally near each other. It also puts some restrictions on the format of each data object which gives us additional bonuses:
  - We can carry out semi-automatic compatibility conversion between different versions of the data model.
  - We can benefit from automated correctness checks for our object.

It is important to note that you can easily start using Active Data with absolutely no knowledge on the OCAF itself. However, those people who start using pure OCAF often eventually end up (after spending many days of work) with something very similar to Active Data framework. Therefore, we suggest AD library as a natural crystallization of chaotic OCAF experience. Active Data brings into creative OCAF mystery some boring order and noticeably reduces the resulting entropy especially because of comprehensive unit tests. More than 100 test cases are available for typical operations (like deletion, copy/paste) and this number continuously grows. It is worth saying that most tests are not of simple getter/setter style: sometimes they are very complicated.

Of course, Active Data is not a panacea for any data design problem. However, its generality and usability was proved by usage in real industrial software. Moreover, AD is an open source project which makes you feel confident about the future of your own product: source code, unit tests and documentation are at your full disposal.

This documentation describes the principles of AD library in the following sections:

Check out appendices:

Chapter 2

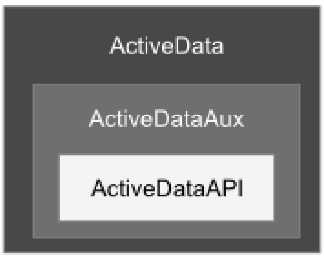
Getting Started

2.1 What is in the distribution?

Active Data is shipped as a collection of six libraries:

Library	Purpose
`ActiveDataAPI`	Contains pure abstract classes (interfaces) for the main data types and utilities of the library. Think of this package as of an entry point to the Active Data basics.
`ActiveDataAux`	Some useful utilities which are not related to OCAF and data framework but intensively used internally.
`ActiveData`	Core classes representing the atomic blocks for constructing your data objects. This package also contains the algorithmic kernel of the framework (dependency graphs, undo/redo, copy/paste etc).
`ActiveDataTestLib`	Test engine. Contains launching mechanism for unit tests, processor of description files, report generator etc.
`ActiveDataTest`	Actual test cases. Normally these tests are launched automatically after compilation of AD libraries.
`ActiveDataDraw`	Draw extensions for working with Active Data via command line.

As a software developer you are normally interested in only three libraries doing all the job: `ActiveDataAPI`, `ActiveDataAux` and `ActiveData`:



API and Aux packages are very small and you can wonder if we really need to have them separated. Well, the possibility to merge them is really considered for the future (check out the room for improvement section), but we have to admit that such separation is quite convenient. Indeed, API gives you a list of main entities you are supposed to work with. Aux package has nothing to do with OCAF and persistence, but provides some utilities which are normally revolving around data models (e.g. abstract mechanism of message logging, expression evaluation etc).

2.2 Dependencies from 3-rd parties

The dependencies of Active Data library from the 3-rd party products are reduced to essential minimum. The required products are listed in the following table:

3-rd party	Why do I need this stuff?
`Open CASCADE Technology (OCCT)`	As we already mentioned above, AD is based on OCAF. That is why OCCT is the essential groundwork for Active Data. Moreover, our framework takes advantage of the entire OCCT ecosystem: we use the mechanism of smart pointers, standard OCCT collections and plenty of different tools.
`Intel TBB`	This is the way how CPU-based parallelism is injected into AD. Basically, this dependency is not very strict and can be avoided. However, we keep it essential in the standard delivery as TBB is widely used in OCCT itself. Therefore, normally there is no problem to have TBB plugged. At the price of introducing the additional 3-rd party, we benefit from the friendly and powerful parallelism capabilities of TBB.

2.3 Overview

Active Data framework provides a toolkit for creation of custom data models based on OCAF. This toolkit contains a set of basic classes and common mechanisms for connecting objects into hierarchical structures and binding algorithms with them (if necessary). From the very general point of view, the data model implemented with help of Active Data framework represents an *interface* to the underlying OCAF structures. We use the idea (pattern) of *data access object* which is popular in enterprise software domain dealing with databases. It is easy to draw a parallel between OCAF and a database, so that the concept of data access object is just the same. This object does not contain your real data. It rather allows getting access to your data stored aside.

AD data access objects are called *Data Cursors:* this way we emphasize the volatile nature of these objects. The Data Cursors are classified by their locality as described in the following table:

Data Cursor type	Purpose
`ActAPI_IModel`	The biggest Cursor. Normally you have only one Model instance for your data. It covers the entire project hierarchy.
`ActAPI_IPartition`	Interface for accessing a "folder" containing data objects of the same type. Using Partitions it is very easy to iterate over the objects of a specific type in their actual storage order (used internally by OCAF).

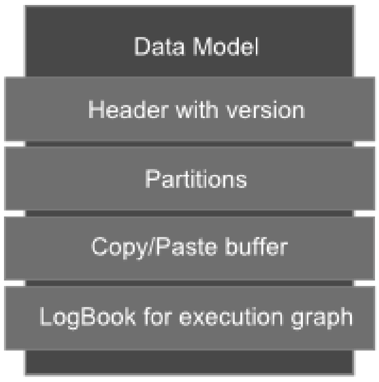
`ActAPI_INode`	Accessor to a data object. It should be understood from the very beginning that *your job as a developer is to design your data objects*. Unlike doing this with pure OCAF (where you have a complete freedom in assembling the so called labels and attributes), in Active Data you are a bit more restricted. The main rule is that your data object should be expressed as a collection of primitive-type properties called *Parameters*.
`ActAPI_IParameter`	Accessor to a primitive-type property of the data object. In Active Data framework this is the terminal type of entity: you are not supposed to deal with any entities which are "more atomic" than Parameters.

Keeping the internal OCAF tree in mind, you can think of a Data Cursor as of a "sliding window" moving on the surface of labyrinthine OCAF substance. Such architecture allows you constructing the data access objects by demand, once you really need to work with the persistent data. Hereinafter we discuss this concept more precisely.

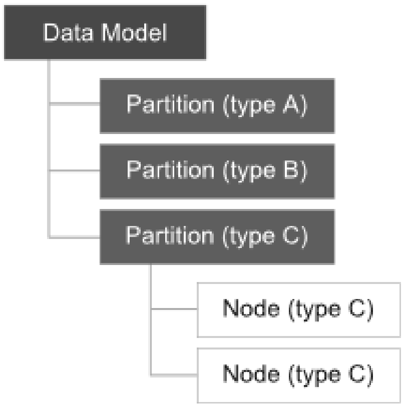
2.3.1 Structure of Model

Active Data uses the following main object types:

- Data Model: the root object playing as an entry point to the entire data. A single Model object owns one or several Partition instances depending on the number of Node types in your application. Each Partition contains Nodes of a particular type. Such structure represents an internal data tree: it is not supposed to be shown to the user;
- Partition: dedicated abstraction to group Nodes of a particular type together. Such distribution of Nodes by the corresponding Partitions allows us having well-structured internal OCAF tree where different types of Nodes are not mixed up. Physically the data model structure is a single-level list of Partitions, each containing a set of Nodes of a certain type;
- Node: the main abstraction representing an interface to the domain-specific data object. Internally each Node is a composition of properties and relations between them. Each such property and relation is represented by a Parameter object.
- Parameter: primitive portion of data or relation attribute associated with the Node. Active Data provides a set of ready-to-use Parameters for primitive data types, such as Integer, Real, String, Unicode String, OCCT Shape etc. Relations are represented by Tree Functions, Tree Nodes and general-purpose References;
- Tree Function: the abstraction for a parametric dependency in the data model between an arbitrary set of input Parameters and an arbitrary set of output Parameters. A Tree Function is responsible for updating the output Parameters using the input Parameters and some application-specific algorithm. All Tree Functions defined by the end-user application are invoked automatically by the framework each time the update of the data model is requested. Tree Functions are identified with global unique IDs.



For daily development it is enough to keep in mind the following structure:



2.3.2 Structure of Nodes

The data structures based on OCAF are normally quite sophisticated in their internal organization. Without a super-layer like Active Data or any other convenience framework it is not easy to operate with your persistent objects. Data Node is a concept giving you a higher level of abstraction over the unfriendly OCAF. As we already said, a Data Node can be seen as a sliding window moved in a space of raw OCAF data and settled down on those portions of the internal hierarchy (`TDF_Label` entities) which you want to *interpret as data objects*. A single Data Node corresponds to a single object.

Note

Most often in practice there is no sense to distinguish between Data Nodes and the underlying data objects. Even though it is possible to charge a single Node consequently with several objects, normally you do not proceed like this. It is programmatically more convenient just to recreate a Data Node in each working session with your data object. As Data Nodes are manipulated by smart pointers, they will die automatically once they are not used any longer. Just notice that destruction of a Data Node does not mean destruction of a data object. The latter will be kept in your project regardless of the lifetime of your Data Cursors.

If we recall that a Data Node is a concrete realization of the data access object pattern, it will not be surprising that it has three possible states:

- DETACHED: the Data Node is not bound to any raw data (`TDF_Label`) and cannot be used for data manipulation so;
- ATTACHED + WELL-FORMED: the Data Node is charged with valid raw data and can be used for data manipulation so;
- ATTACHED + BAD-FORMED: the Data Node is charged with invalid raw data. This may happen if the Data Cursor is moved to improper OCAF label or if the underlying data object is not yet initialized. This state of a

Data Node gives you a limited access to the underlying data. E.g. you can populate the underlying OCAF and switch your cursor to a WELL-FORMED state so, but you cannot read the underlying data until you perform such population.

As mentioned above, each Data Node is represented by a set of Parameters. Just like Data Nodes, the Parameters also follow the Data Cursor ideology. They are designed as object-oriented accessors to primitive properties of your data objects. Each Node contains two groups of Parameters:

- Internal Parameters;
- Custom Parameters.

Internal Parameters represent some technical aspects of the framework itself. They exist for any kind of Data Node forming up its immutable kernel. For instance, internal Parameters are used to store the object type, parent-child relations, back-references etc. Normally you never deal with these internal Parameters directly and affect them only implicitly (e.g. when you create a reference, the back-reference appears automatically).

Normally you are always focused on the *custom* Parameters which are stored separately from the internal ones. Actually, the overall data model design is reduced to the following activities:

- Design your Data Node classes, specifying which Parameters exactly your objects contain. One object type corresponds to one Data Node class which can deal with it;
- Think over the relations between your data objects and configure these relations using the convenience methods of each Node. E.g. you can add one object as a child of another one.

2.3.3 Parameters

This section describes the set of pre-implemented Parameters which can be (and supposed to be) used for composition of your custom data objects. Each Parameter is a convenient wrapper under some OCAF type (`TDF_Attribute`). A Parameter contains only the primitive setter and getter logic which allows transferring data to/from OCAF following the data access object ideology. The following Parameters are shipped with the framework:

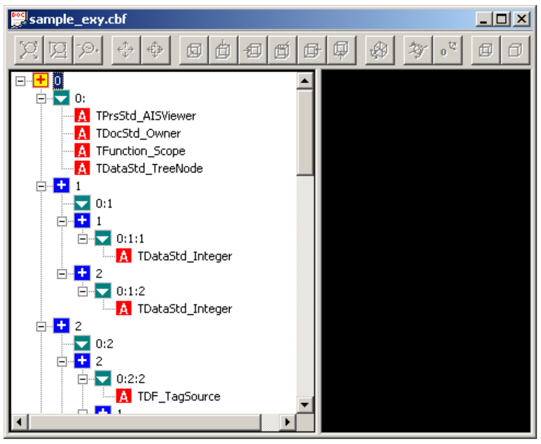
Parameter type	Description
`ActData_AsciiStringParameter`	Represents ASCII string property. A Node can contain any number of such Parameters.
`ActData_StringArrayParameter`	Represents one-dimensional array of ASCII strings. A Node can contain any number of such Parameters.
`ActData_StringMatrixParameter`	Represents two-dimensional array of ASCII strings. A Node can contain any number of such Parameters.
`ActData_BoolParameter`	Represents Boolean value. A Node can contain any number of such Parameters.
`ActData_BoolArrayParameter`	Represents one-dimensional array of Boolean values. A Node can contain any number of such Parameters.

`ActData_BoolMatrixParameter`	Represents two-dimensional array of Boolean values. A Node can contain any number of such Parameters.
`ActData_IntParameter`	Represents integer value. A Node can contain any number of such Parameters.
`ActData_IntArrayParameter`	Represents one-dimensional array of integer values. A Node can contain any number of such Parameters.
`ActData_IntMatrixParameter`	Represents two-dimensional array of integer values. A Node can contain any number of such Parameters.
`ActData_RealParameter`	Represents floating-point (double precision) value. A Node can contain any number of such Parameters.
`ActData_RealArrayParameter`	Represents one-dimensional array of floating-point values. A Node can contain any number of such Parameters.
`ActData_RealMatrixParameter`	Represents two-dimensional array of floating-point values. A Node can contain any number of such Parameters.
`ActData_GroupParameter`	Represents a sort of "header" for each logical group of Parameters. Group Parameter does not have any value. A Node can contain any number of such Parameters.
`ActData_ShapeParameter`	Represents CAD shape. A Node can contain any number of such Parameters.
`ActData_MeshParameter`	Represents mesh data. This is the only non-standard data type implemented in AD framework comparing to the pure OCAF. A Node can contain any number of such Parameters.
`ActData_SelectionParameter`	Represents a mask of integer IDs. You can think of this Parameter type as of a bitset. A Node can contain any number of such Parameters.
`ActData_NameParameter`	Represents a Unicode string. A Node can contain any number of such Parameters.
`ActData_TypeNameParameter`	Represents an ASCII string for Node type. A Node can contain only a single instance of this Parameter.
`ActData_TreeNodeParameter`	Relational Parameter which allows creation of a parent-child hierarchy of Nodes. This hierarchy normally represents a user-tree of Nodes which is supposed to be shown in GUI. A Node can contain only a single instance of this Parameter.

`ActData_TreeFunctionParameter`	Relational Parameter which allows creation of a dependency graph between arbitrary Parameters. This graph is traversed by execution mechanism in order to perform automatic re-calculation of dependent values with custom algorithms. A Node can contain any number of such Parameters.
`ActData_ReferenceParameter`	Represents a reference to an arbitrary data object (or its property). This Parameter type is used to establish general-purpose relations within your model. A Node can contain any number of such Parameters.
`ActData_ReferenceListParameter`	Represents a reference list to arbitrary data objects (or their properties). This Parameter type is used to establish a set of general-purpose relations within your model. A Node can contain any number of such Parameters.

It should be clear now that using Active Data you have to express your custom data objects using only a limited set of Parameters. It is known from practice that the available Parameter types are enough to construct even very sophisticated objects. However, there are some key points to keep in mind when designing your data model:

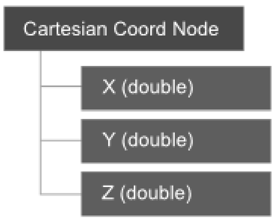
- All Parameters corresponding to the collections (except only the Reference List Parameter) are of static sizes. The lack of dynamically growing collections is a serious limitation, but it can be easily compensated by the fact, that you can dynamically link your data objects to each other. Each time you need to have a non-static collection, consider using auxiliary objects (most likely hidden for the end-users) for solving this problem;
- Normally people who use OCAF do not neglect the possibility to introduce their custom attribute (`TDF_Attribute`) types. The latter may seem convenient, because this way you do not have to decompose your tricky object onto the standard primitives available in OCAF. We strongly discourage this practice and do not allow extension of available Parameters due to the following reasons:
  - Just like everything in the digit world can be expressed with 1 and 0, nothing prevents you from expressing your sophisticated structures with available set of Parameters. Sometimes this implies construction of hidden Nodes and sub-Nodes;
  - Each non-standard Parameter decreases compatibility of your Data Model with the standard OCAF format. Currently (if we exclude Mesh Parameter from consideration) it is possible to read Active Data documents using common tools available for OCAF. This generality is very valuable in our vision. It not only gives you possibility to use all common OCAF utilities (e.g. for browsing your model), but also fits into the primary philosophy of AD framework: *to be an OCAF-usage practice, rather than OCAF-using product.*



Looking inside a Parameter, you will shortly notice that besides its name, type and value, it can also store something called a "semantic ID". The latter concept is very useful in engineering applications when you treat your Parameters as variables having some physical nature. In that case the semantic ID could be a reference to the measurement units, range of validity or whatever else. It is a job of a concrete software to properly handle these semantic IDs.

2.3.4 Example

Now let us design a simple data object having the following structure:



It is a single point in a three-dimensional space given its Cartesian coordinates. On this simple case we will study how to design the corresponding data object using Active Data. We start with the real OCAF source corresponding to this object in AD:

```
ACT Data Model dump. Bound FRAMEWORK version: 1.0.0
ACT Data Model dump. Bound APPLICATION version: 1.0.0
L >> [0:]
A >> [TDocStd_Owner] - ### cannot dump ###
A >> [TDataStd_TreeNode] - ### cannot dump ###
A >> [TFunction_Scope] - ### cannot dump ###
A >> [TNaming_UsedShapes] - ### cannot dump ###
L >> [0:1]
     L >> [0:1:1]
     A >> [TDataStd_Integer] - 65536
     L >> [0:1:2]
     A >> [TDataStd_Integer] - 65536
====================================================================================================
+-> [0:2:10] PARTITION [ABCData_XYZPartition]
     L >> [0:2:10]
     A >> [TDF_TagSource] - ### cannot dump ###
     L >> [0:2:10:1]
    ************************************************************************************
+-> [0:2:10:1] NODE [ABCData_XYZNode] -
    ************************************************************************************
          L >> [0:2:10:1:1]
               L >> [0:2:10:1:1:1]
                   -------------------------------------------------------------
                    [0:2:10:1:1:1] PARAMETER [ActData_TypeNameParameter]
                   -------------------------------------------------------------
                    L >> [0:2:10:1:1:1:1]
                    A >> [TDataStd_Integer {Type}] - 1024
                    L >> <MTime is not dumped>: tag = 5
                    L >> [0:2:10:1:1:1:6]
                    A >> [TDataStd_Integer {Validity}] - 1
                    L >> [0:2:10:1:1:1:8]
                    A >> [TDataStd_Integer {Pending}] - 0
                    L >> [0:2:10:1:1:1:101]
                    A >> [TDataStd_AsciiString] - ABCData_XYZNode
               L >> [0:2:10:1:1:2]
                   -------------------------------------------------------------
                    [0:2:10:1:1:2] PARAMETER [ActData_TreeNodeParameter]
                   -------------------------------------------------------------
                    L >> [0:2:10:1:1:2:1]
                    A >> [TDataStd_Integer {Type}] - 128
                    L >> <MTime is not dumped>: tag = 5
                    L >> [0:2:10:1:1:2:6]
                    A >> [TDataStd_Integer {Validity}] - 1
                    L >> [0:2:10:1:1:2:8]
                    A >> [TDataStd_Integer {Pending}] - 0
                    L >> [0:2:10:1:1:2:101]
                    A >> [TDataStd_TreeNode] - ### cannot dump ###
               L >> [0:2:10:1:1:3]
```

```
        ------------------------------------------------------------
         [0:2:10:1:1:3] PARAMETER [ActData_ReferenceListParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:3:1]
         A >> [TDataStd_Integer {Type}] - 524288
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:3:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:3:8]
         A >> [TDataStd_Integer {Pending}] - 0
    L >> [0:2:10:1:1:4]
        ------------------------------------------------------------
         [0:2:10:1:1:4] PARAMETER [ActData_ReferenceListParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:4:1]
         A >> [TDataStd_Integer {Type}] - 524288
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:4:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:4:8]
         A >> [TDataStd_Integer {Pending}] - 0
    L >> [0:2:10:1:1:5]
        ------------------------------------------------------------
         [0:2:10:1:1:5] PARAMETER [ActData_ReferenceListParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:5:1]
         A >> [TDataStd_Integer {Type}] - 524288
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:5:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:5:8]
         A >> [TDataStd_Integer {Pending}] - 0
         L >> [0:2:10:1:1:5:101]
         A >> [TDataStd_ReferenceList] - {0:2:5:1:2:109}
    L >> [0:2:10:1:1:6]
        ------------------------------------------------------------
         [0:2:10:1:1:6] PARAMETER [ActData_IntParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:6:1]
         A >> [TDataStd_Integer {Type}] - 1
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:6:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:6:8]
         A >> [TDataStd_Integer {Pending}] - 0
         L >> [0:2:10:1:1:6:101]
         A >> [TDataStd_Integer] - 13
    L >> [0:2:10:1:1:7]
        ------------------------------------------------------------
         [0:2:10:1:1:7] PARAMETER [ActData_TreeFunctionParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:7:1]
         A >> [TDataStd_Integer {Type}] - 64
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:7:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:7:8]
         A >> [TDataStd_Integer {Pending}] - 0
    L >> [0:2:10:1:1:8]
        ------------------------------------------------------------
         [0:2:10:1:1:8] PARAMETER [ActData_TreeFunctionParameter]
        ------------------------------------------------------------
         L >> [0:2:10:1:1:8:1]
         A >> [TDataStd_Integer {Type}] - 64
         L >> <MTime is not dumped>: tag = 5
         L >> [0:2:10:1:1:8:6]
         A >> [TDataStd_Integer {Validity}] - 1
         L >> [0:2:10:1:1:8:8]
         A >> [TDataStd_Integer {Pending}] - 0
    L >> [0:2:10:1:1:9]
        ------------------------------------------------------------
         [0:2:10:1:1:9] PARAMETER [ActData_TreeFunctionParameter]
        ------------------------------------------------------------
```

```
                L >> [0:2:10:1:1:9:1]
                A >> [TDataStd_Integer {Type}] - 64
                L >> <MTime is not dumped>: tag = 5
                L >> [0:2:10:1:1:9:6]
                A >> [TDataStd_Integer {Validity}] - 1
                L >> [0:2:10:1:1:9:8]
                A >> [TDataStd_Integer {Pending}] - 0
      L >> [0:2:10:1:2]
           L >> [0:2:10:1:2:100]
               ------------------------------------------------------------
                [0:2:10:1:2:100] PARAMETER [ActData_NameParameter]
               ------------------------------------------------------------
           A >> [TDataStd_Name] - Point 1
                L >> [0:2:10:1:2:100:1]
                A >> [TDataStd_Integer {Type}] - 512
                L >> <MTime is not dumped>: tag = 5
                L >> [0:2:10:1:2:100:6]
                A >> [TDataStd_Integer {Validity}] - 1
                L >> [0:2:10:1:2:100:8]
                A >> [TDataStd_Integer {Pending}] - 0
           L >> [0:2:10:1:2:101]
               ------------------------------------------------------------
                [0:2:10:1:2:101] PARAMETER [ActData_RealParameter]
               ------------------------------------------------------------
                L >> [0:2:10:1:2:101:1]
                A >> [TDataStd_Integer {Type}] - 2
                L >> [0:2:10:1:2:101:2]
                A >> [TDataStd_Name {Name}] - X
                L >> <MTime is not dumped>: tag = 5
                L >> [0:2:10:1:2:101:6]
                A >> [TDataStd_Integer {Validity}] - 1
                L >> [0:2:10:1:2:101:7]
                A >> [TDataStd_Integer {UFlags}] - 0
                L >> [0:2:10:1:2:101:8]
                A >> [TDataStd_Integer {Pending}] - 0
                L >> [0:2:10:1:2:101:101]
                A >> [TDataStd_Real] - 0
           L >> [0:2:10:1:2:102]
               ------------------------------------------------------------
                [0:2:10:1:2:102] PARAMETER [ActData_RealParameter]
               ------------------------------------------------------------
                L >> [0:2:10:1:2:102:1]
                A >> [TDataStd_Integer {Type}] - 2
                L >> [0:2:10:1:2:102:2]
                A >> [TDataStd_Name {Name}] - Y
                L >> <MTime is not dumped>: tag = 5
                L >> [0:2:10:1:2:102:6]
                A >> [TDataStd_Integer {Validity}] - 1
                L >> [0:2:10:1:2:102:7]
                A >> [TDataStd_Integer {UFlags}] - 0
                L >> [0:2:10:1:2:102:8]
                A >> [TDataStd_Integer {Pending}] - 0
                L >> [0:2:10:1:2:102:101]
                A >> [TDataStd_Real] - 0
           L >> [0:2:10:1:2:103]
               ------------------------------------------------------------
                [0:2:10:1:2:103] PARAMETER [ActData_RealParameter]
               ------------------------------------------------------------
                L >> [0:2:10:1:2:103:1]
                A >> [TDataStd_Integer {Type}] - 2
                L >> [0:2:10:1:2:103:2]
                A >> [TDataStd_Name {Name}] - Z
                L >> <MTime is not dumped>: tag = 5
                L >> [0:2:10:1:2:103:6]
                A >> [TDataStd_Integer {Validity}] - 1
                L >> [0:2:10:1:2:103:7]
                A >> [TDataStd_Integer {UFlags}] - 0
                L >> [0:2:10:1:2:103:8]
                A >> [TDataStd_Integer {Pending}] - 0
                L >> [0:2:10:1:2:103:101]
                A >> [TDataStd_Real] - 0
```

Here `L` denotes `TDF_Label` and `A` — `TDF_Attribute`.

Note

> In order to understand the listing above, you have to possess the basic knowledge of OCAF. If you do not feel confident in this area, just skip this part of discussion: understanding the provided information is not required for successful usage of AD framework.

At a first glance we see that in order to have just three floating-point values persistent, we are forced to occupy much more additional space for different environmental stuff. In this regard we recommend you never to express a cloud of N points with N data objects. Doing so you will sacrifice a lot of memory to internal technical attributes of the framework. You would better store all coordinates in a single array or use three arrays for X, Y and Z coordinates respectively. Now let us take a closer look on the contents of this listing.

The first block in the given listing corresponds to the version header. There are two version numbers (both equal to `1.0.0` in our sample) associated with the AD-based data model: one is the version of AD framework, another — the version of your application. These two version numbers can be used for compatibility conversion.

```
ACT Data Model dump. Bound FRAMEWORK version: 1.0.0
ACT Data Model dump. Bound APPLICATION version: 1.0.0
L >> [0:]
A >> [TDocStd_Owner] - ### cannot dump ###
A >> [TDataStd_TreeNode] - ### cannot dump ###
A >> [TFunction_Scope] - ### cannot dump ###
A >> [TNaming_UsedShapes] - ### cannot dump ###
L >> [0:1]
     L >> [0:1:1]
     A >> [TDataStd_Integer] - 65536
     L >> [0:1:2]
     A >> [TDataStd_Integer] - 65536
```

Note also that there is a bunch of standard attributes like `TDocStd_Owner`, `TDataStd_TreeNode`, `T-Function_Scope` and `TNaming_UsedShapes`. We do not want to start a detailed discussion of these attributes as we are limited in paper here. Just keep in mind that being associated to the root label, these attributes play an essential role in forming of a valid document.

Starting from `[0:2:10]` entry the section with Partitions follows. It contains the XYZ data objects as children. It should be noted that this parent-child relationship is only a technical (storage) hierarchy, so it is not supposed to be shown to the user. In order to produce an alternative (user-oriented) tree the `ActData_TreeNodeParameter` has to be used. We have to pay a special attention to sections `[0:2:10:1:1]` and `[0:2:10:1:2]` under the data object `[0:2:10:1]`. The first section contains the internal properties which are not accessible for application developer and used by the framework itself (type information, back-references, optional evaluator functions etc). The second section with entry `[0:2:10:1:2]` is a container of custom properties designed by the application developer.

Now let us take a look on the class definition for `ABCData_XYZNode` object. We refer you to the unit tests of AD framework for live exercises and limit our discussion here with only a brief overview of Data Node class.

```
DEFINE_STANDARD_HANDLE(ABCData_XYZNode, ActData_BaseNode)

class ABCData_XYZNode : public ActData_BaseNode
{
public:

  //! IDs of the involved Parameters.
  enum ParamId
  {
    PID_PointName = ActData_BaseNode::UserParam_Last,
    PID_X,
    PID_Y,
    PID_Z,
    PID_Last = ActData_BaseNode::UserParam_Last + ActData_BaseNode::RESERVED_PARAM_RANGE
  };

public:

  DEFINE_STANDARD_RTTI(ABCData_XYZNode, ActData_BaseNode) // OCCT RTTI
  DEFINE_NODE_FACTORY(ABCData_XYZNode, Instance)
public:

  ABCData_EXPORT static Handle(ActAPI_INode) Instance();

// Generic naming support:
public:

  ABCData_EXPORT virtual TCollection_ExtendedString
    GetName();

  ABCData_EXPORT virtual void
    SetName(const TCollection_ExtendedString& theName);

// Initialization:
public:

  ABCData_EXPORT void
    Init();

private:

  //! Allocation is allowed only via Instance method.
  ABCData_XYZNode();

};
```

In this header file we declare `ABCData_XYZNode` as a subclass of `ActData_BaseNode`. For people familiar with Open CASCADE Technology it is not surprising to see `DEFINE_STANDARD_HANDLE` definition which binds a smart pointer for our Data Node.

Note

> Theoretically, it is not required to use smart pointers for the Data Cursors. However, we do so as the basic `ActAPI_IDataCursor` class inherits `Standard_Transient` designed for reference counting. Moreover, AD framework uses these smart pointers very intensively.

Then the enumeration with Parameter IDs follows:

```
enum ParamId
{
  PID_PointName = ActData_BaseNode::UserParam_Last,
  PID_X,
  PID_Y,
  PID_Z,
  PID_Last = ActData_BaseNode::UserParam_Last + ActData_BaseNode::RESERVED_PARAM_RANGE
};
```

This enumeration declares the integer tags which will be used for the root labels of the corresponding Parameters. If you do not care of the internal OCAF peculiarities, just think of these numbers as of unique identifiers to address your atomic Parameters. However, this enumeration is not enough to have your Data Node completely defined. In order to finish Parameters definition, you have to implement the `ABCData_XYZNode()` constructor using the following magic macro:

```
//! Default constructor. Registers the internal set of Parameters.
ABCData_XYZNode::ABCData_XYZNode() : ActData_BaseNode()
{
  REGISTER_PARAMETER     (Name, PID_PointName);
  REGISTER_PARAMETER_EXPR(Real, PID_X);
  REGISTER_PARAMETER_EXPR(Real, PID_Y);
  REGISTER_PARAMETER_EXPR(Real, PID_Z);
}
```

Notice that there are two kinds of macro available:

- `REGISTER_PARAMETER`: simply associates your ID with the given Parameter type;

- `REGISTER_PARAMETER_EXPR`: does the same, but also binds an expression evaluation mechanism to the corresponding property. The latter mechanism is designed for automatic calculation of your numerical (but not only) properties using an abstract expression evaluator (see `ActData_RealEvalFunc` for example). If you do not need this kind of parametric dependencies in your data model, use simple `REGISTER_PARAMETER` macro.

Construction of the Data Node is done via factory method `Instance()`. This rule has to be respected in order to allow the framework automatic creation of your Data Nodes (the latter is achieved thanks to `DEFINE_NODE_FACTORY` macro).

For the complete example on designing your custom data model, please, refer to `ActiveDataTest` library (sample classes can be found in `/Data/TestModel` directory).

Chapter 3

Tree Functions

3.1 Overview

If you ever worked with relational databases, then the notion of a *stored procedure* should be familiar to you. If not, then there are just few things to keep in mind about these monsters:

- Stored procedure is an *algorithm stored together with data* (at the same architectural level).
- Stored procedure can simplify the logic of your software: some part of this logic is transferred from your code to the database.

The reasons to keep some algorithms right at the database level could be different. In client-server architectures we can figure out the followings:

- Performance: stored procedures can accelerate your calculations.
- Architecture: sometimes it is just more natural to have a stored procedure instead of a classical algorithm.
- Observer: necessity to react automatically on data modification events (what is known as "trigger" concept).

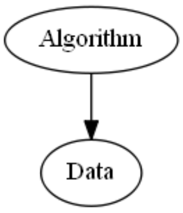
In OCAF it is possible to have a similar sort of *persistent algorithms.* There are some differences in the conception as we are speaking about desktop applications without any client-server architecture inside. Therefore, performance is basically not an issue as everything is operating on a local station. Likewise, there is no concurrent access to the data by several consumers and no protection or locking is required so. You should normally consider usage of persistent algorithms in the following situations:

- You want to organize your hierarchical data into dependency graphs. This way you establish one more relationship between your data objects expressing the computational graph.
- You want to re-execute automatically all dependent algorithms once their inputs are changed.

Injecting this mechanism makes your data model more intelligent regarding the process of data update. Doing so, you only need to build your dependency graph (you may also rebuild it dynamically according to some rules) and implement each algorithm as a node in this graph. Of course, such a mechanism is not always something what you *need* to have. It fits best those engineering problems where calculations can be naturally organized into the graph structures (CAE, parametric CAD, CAD optimization, etc.).

3.2 Architecture

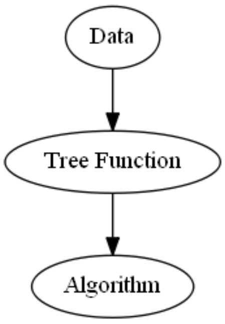
It should be clearly understood that having or not the execution graphs in your data model is a major architectural question which you have to decide on your own. Let us consider a classical situation when you build up your algorithms around the *passive* bank of data (no persistent algorithms are used). In that case the algorithm is typically aware of the data model and can manipulate with the data objects directly.



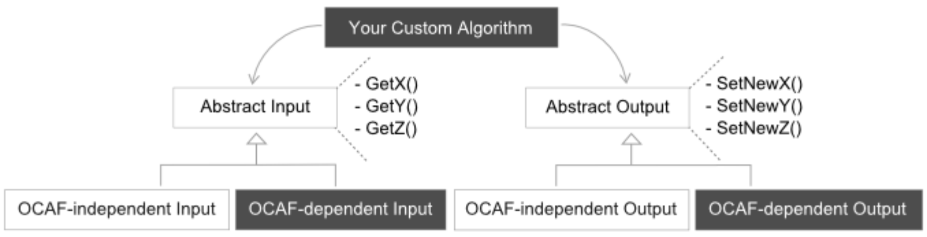
Let us now consider a kind of *active* data model. In that case the data model maintains a graph of algorithmic dependencies. It also takes responsibility of supplying the algorithms with inputs and updating their outputs once computation is done. In AD framework, any modification of the input data leads to invocation of a conventional `Execute()` method of a domain-specific class inheriting from `ActData_BaseTreeFunction`. Normally you need to have a number of Tree Function classes dealing with your custom logic inside their `Execute()` methods. At a first glance it looks like the algorithmic stuff is mixed up with the data model internals.

Basically, you are not restricted to put your entire algorithm possessing, for example, 1 billion instructions right into `Execute()` method. It will work, but from architectural point of view this is pure evil. The more graceful decision is to separate your algorithms from the data model conceptually (e.g. put them into different libraries). Doing so, you decouple your domain-specific "business logic" from the data organization format. Such a solution gives you the following advantages:

- You may change your data model without affecting the algorithms;
- You may test your algorithms without necessity to build up a sample data model;
- You may use your algorithms outside your software as a usual library.



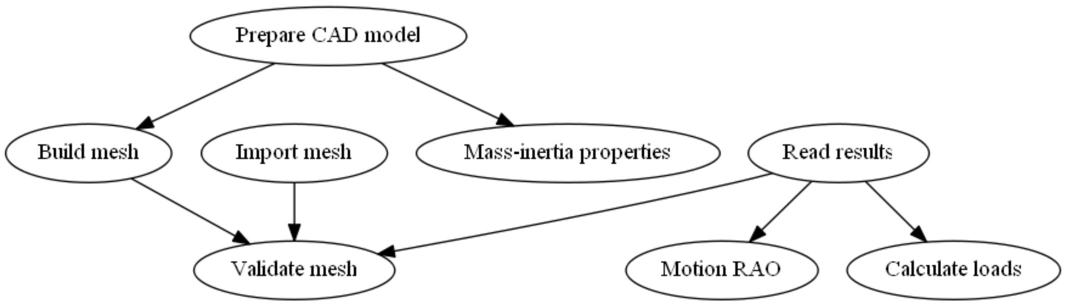
In order to break the link from an algorithm to a data model, a well-known *Adapter* pattern can be employed. The idea is very simple: you build a thin "bridge" from your data model to an algorithm using a specific realization of an abstract adapter class. The algorithm in its turn works with its inputs and outputs via the corresponding adapters only. The deal is better clarified by the following picture:



Here *Your Custom Algorithm* uses *Abstract Input* and *Abstract Output* as adapters for input and output data. The algorithm knows nothing about the data model and can operate with only those information available via adapters.

In our sample it can get three floating-point values X, Y and Z by means of `GetX()`, `GetY()` and `GetZ()` methods. Internally it will do something with the input values and then it will set the calculation results to an abstract output by means of `SetNewX()`, `SetNewY()` and `SetNewZ()` methods. There is no word about OCAF and the actual data model, so your algorithm comes with the pure domain-specific logic. The real connection of your data model with the algorithm is achieved by extending the abstract adapter classes and implementing the declared interface methods.

Wrapping your algorithms with Tree Functions and configuring their dependencies, you obtain an execution graph as a model of your calculation network. The picture below gives an exemplary graph for some hypothetical calculations in hydrodynamics domain:



Let us comment a bit on the provided sample. The first "Prepare CAD model" function is used as a source of CAD geometry. It can read the model from file or build it from scratch. Once the CAD model is ready, we have several jobs to do: to build a mesh for subsequent analysis (e.g. CFD or diffraction/radiation) or to calculate mass-inertia properties of our geometry. The directed links between the parent node and these two child functions mean that the latter functions have some inputs in common with the outputs of "Prepare CAD model". Another way to get a mesh is to import it from file. The latter possibility is represented by "Import mesh" function. Whatever approach for meshing you use, it will end up in "Validate mesh" function which will check the correctness of your panels before the real hydrodynamics analysis is launched. We skip the rest of dependencies illustrated in the given graph as the main idea should be already clear enough.

Your job as a developer is to feed each function with its direct input and output Parameters. You do not care of how these Parameters are traversed in order to build up the execution graph. The latter is a job of Active Data.

3.3 Variables

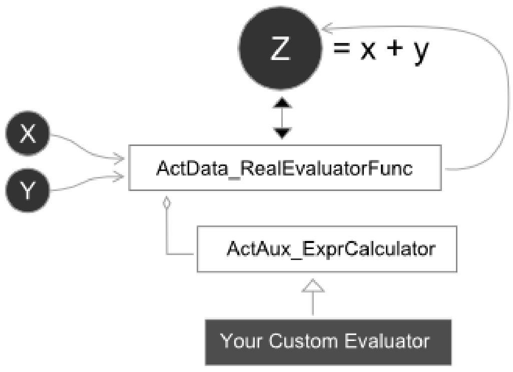
Active Data comes with a specialization of Tree Function mechanism for implementation of global variables. By a variable we understand a specific data object having the following properties:

- Name;
- Value;
- Semantic ID;
- Evaluation string (expression).

Semantic ID is purposed to address the real physical nature of your variable. Using it, you can associate with your variable the measurement units, validity range or whatever else. Active Data does not provide any concrete solutions for semantic IDs as any such solution will be too domain-specific. One good practice is to prepare a kind of XML dictionary enumerating all useful physical quantities employed in your software. Then it is natural to bind an ID of the proper XML entry as a semantic ID of your variable.

Evaluation string gives a mathematical expression to be parsed and evaluated in order to calculate the value of your variable. It should contain only those operands which correspond to other variables in your project. Active Data does not attempt to specialize syntax for these mathematical expressions as well as it does not come with a concrete evaluator. You may use any 3-rd party or your own evaluation tool for processing these expressions. E.g. it is customary to use interpreters like Python or Tcl.

The following variable types are currently supported: integer, real, bool and string. However, technically there is no problem to introduce other variable types. The only challenge is how are you going to evaluate them. Currently evaluation mechanism is available for floating-point variables only. In order to start using it, you may take advantage of the standard `ActData_RealEvaluatorFunc` Tree Function.



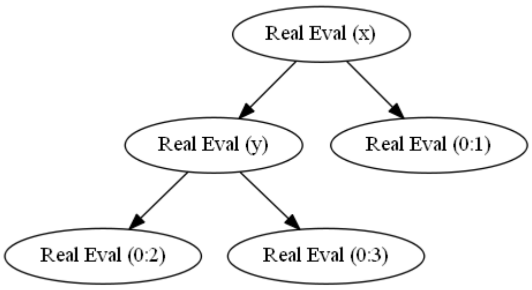
In order to connect variables with the Parameters of your Nodes, you have to register your Parameters as "express-ible". This is done in the constructor of your Node by means of a convenient macro.

```
//! Default constructor.
ActTest_StubANode::ActTest_StubANode()
{
  REGISTER_PARAMETER     (Name,         PID_Name);
  REGISTER_PARAMETER     (Shape,        PID_DummyShapeA);
  REGISTER_PARAMETER     (Shape,        PID_DummyShapeB);
  REGISTER_PARAMETER_EXPR(Real,         PID_Real); // (!) This Parameter can be evaluated
  REGISTER_PARAMETER     (TreeFunction, PID_TFunc);
  REGISTER_PARAMETER     (Reference,    PID_Ref);
}
```

As the mechanism of variables is nothing but a specialization of Tree Function mechanism, you have to explicitly connect your `ActData_RealEvaluatorFunc` to the proper input and output Parameters. This is done by `ConnectEvaluator()` method of the target Node where the Parameter being evaluated lives.

```
// Real Variable 1 depends on Integer Variables 1 & 2
RealVarNode1->ConnectEvaluator( ActData_RealVarNode::Param_Value,
                                ActAPI_ParameterStream() << IntParam1 << IntParam2 );
```

The dependency graph below illustrates two variables x and y bound with some $y = f(x)$ law. There are also three application-specific Parameters with IDs `0:1`, `0:2` and `0:3` which are evaluated using the declared variables.



There are few service methods at the level of `ActData_BaseModel` which facilitate implementation of a powerful variables mechanism in your application:

- `AddVariable()`: creates a new variable Node in the project;
- `RenameVariable()`: renames the variable with the given ID and adjusts all existing references to this variable in all expressions of your project;

Note

In order to delete variable with all existing references use the standard `DeleteNode()` method.

Summarizing, we would like to underline that introduction of Tree Function mechanism at the data model level makes implementation of variables mechanism easy and natural. We do not discuss the programmatic aspects of its realization as you can find enough examples in the unit tests of Active Data framework: check out `ActTest_-BaseModelEvaluation` test case as reference.

Todo SpyLog functionality still needs to be described.

Chapter 4

Copy & Paste

4.1 Requirements

Note

Copy/Paste functionality provided by Active Data Framework essentially works with Data Nodes and their user-defined sub-trees, i.e. those trees which are built with help of Tree Node Parameters. There is no possibility to copy/paste any individual Parameter without copying the parent Node.

The requirements to Copy/Paste functionality were elaborated basing on the series of formal requirements. We keep this rather historical block here in order to show the original motivation for development of Copy/Paste functionality. The elaborated requirements were formulated as follows:

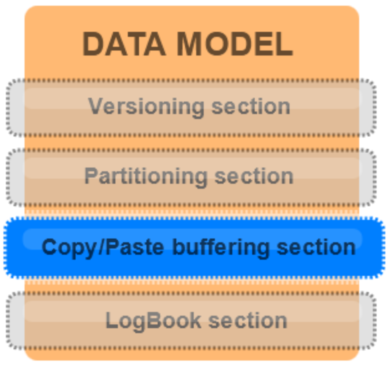
- Copy/Paste functionality must benefit from the standard copy/paste mechanism shipped with original OCAF: we do not want to re-invent the wheel;
- Following the previous requirement, all custom AD-specific OCAF attributes must provide the standardized methods to support copy/paste functionality (i.e. have non-trivial implementation of abstract `TDF_Attribute::Paste()` method). E.g. Mesh Attribute must come with a particular method performing deep copying of the underlying mesh structures;
- Copy/Paste functionality must produce a well-formed resulting sub-tree of data objects. I.e. not only the plain data must be copied, but also all existing references must be normalized in a consistent way. The default behaviour is to ignore those references which point outside the copied structure anyhow. All other references must be adjusted so that to refer to the copy of data instead of the source. Such an approach ensures that the local dependencies are transferred together with their owning sub-trees, while more complex connectivity is just cleaned up in the target copy. This easy-to-remember rule seems to provide a good level of safety in terms of data consistency. On the other hand, such relocation rule might be non-suitable in some specific cases. E.g. we normally want to preserve Parameter-Variable connection when expressible Node is copied. As AD variables mechanism is based on standard Tree Function references, the corresponding Tree Function Parameters will be disconnected regarding to the mentioned rule (Variable Nodes are external to the expressible user-defined Data Nodes). That is why such rule cannot be suggested as general one and must provide customization points for particular use cases. Some examples of such customization points are listed below:
  - User must be free to pass the Tree Function types (i.e. their GUIDs) to the copying routine, enumerating those Tree Functions which have to preserve their global dependencies;
  - User must be free to pass IDs of those Reference Parameters which have to preserve their global pointers after the copy/paste operation completes.
- Sub-tree hierarchy built by means of a Tree Node Parameter must not be broken by the discussed Copy/Paste functionality;

- Copy/Paste functionality must be available as a service sub-set of the `ActiveData` library. For the sake of component separation, the actual implementation of copy/paste mechanism must be packaged into a dedicated auxiliary service class (Copy/Paste Engine) controlled by the Data Model instance;
- User must be able to perform copy/paste operations separately, i.e. in two stages: one is for copying and another — for pasting. If user modifies the source data after he issued the copying request, the result of paste operation should not be affected by such intermediate modifications. I.e. full temporary copy of the source data must be preserved somewhere and be completely independent from the current state of the project tree;
- If the copied data contains a reference to some Node or Parameter which does not exist at the moment of paste operation, the Data Framework should nevertheless allow such pasting and ensure data consistency. The broken (dead) reference should be disconnected from the copied Node. Notice that such "ghost" references can only relate to the data which is external for the copied sub-tree (what we call the out-scoped data). Among all the existing types of references (see below), only direct Reference Parameters and Tree Function Parameters have a chance to pass to the target Node (out-scoped back-references are always lost);
- Pasting into the children of the source Data Node must be prohibited. Pasting into the source Node must be prohibited;
- LogBook must remain unaffected after the copy and paste operations. Otherwise user can experience side effects related to unwanted triggering of Tree Functions on the pasted Parameters. It is important to note that simple cleaning of LogBook cannot be suggested as a common solution as doing so we can loose modification records accumulated out of copy/paste operation. Therefore the copy/paste routine must perform silent modifications (without logging modifications in "push-for-execution" journal) whenever it is possible.

4.2 Copy/Paste Engine

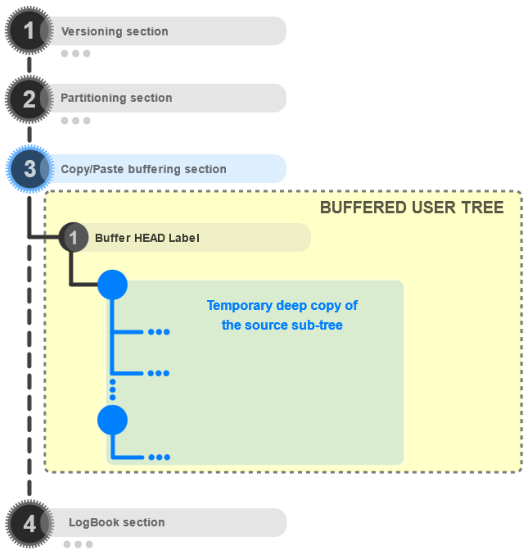
The internal logic of Copy/Paste functionality is implemented in Copy/Paste Engine tool represented by `ActData_CopyPasteEngine` class located in Kernel sub-package of `ActiveData` library. This class is a co-worker one for `ActData_BaseModel` class as it provides the functionality conceptually belonging to the Data Model's responsibility scope. The typical use case for Copy/Paste functionality is as follows:

- User selects a Node to copy (it is called a *source* Node);
- User invokes `ActAPI_IModel::CopyNode()` method which performs full copying of the source data into a temporary place: a dedicated "clipboard" section. This section is very specific as it serves only as a buffer for copy/paste operation. It has its own corresponding OCAF label in the Data Model similarly to what we have for Partitions and version information. The relative ID to access this buffering section is `CopyPasteBuffer` tag available under the root label of the Model;



The following scheme illustrates the actual representation of the copy/paste buffer in the working OCAF document. Notice that the source sub-tree is fully reproduced in the buffering section with the updated references. This is an alternative local hierarchy of Nodes where the root one is detached from any domain-specific parent Data Node.

We use an additional intermediate `Buffer HEAD Label` here for future extensions, e.g. for multiple buffering support.



Note

Notice that such document-level buffering is not conceptually necessary for operations like copy & paste. Indeed, we do need only some transient buffer which remains immutable between the copy and paste requests. Introducing such buffer in the OCAF document can make one think that we are going to persist it which will be obviously redundant. In fact, such a "strange" buffering scheme is justified mainly by our intention to re-use OCAF copy & paste embedded functionality which requires that all the involved labels are attached to the working document. The copy/paste buffer is cleaned up once the model is loaded from a file (if it exists). There is also one little benefit from this approach to buffering: as the temporary copy of data is attached to the same OCAF document as the entire model, we can easily obtain a dump of this buffer via CAF Dumper tool. The latter feature can ease one's life in debugging challenges.

- After such copying is done, user is free to modify the source Node anyhow, e.g. update its Parameters or even delete the Node. All these changes do not affect the temporary copy of data stored in the Copy/Paste Tool, ensuring that the results of paste invocation do not depend on the modifications performed between copy & paste calls;
- User invokes `ActAPI_IModel::PasteAsChild()` method. If no temporary copy exists at the moment, the Data Model class performs no action and returns NULL indicating that invocation logic is wrong. If such copy exists, the Data Model class (actually, its co-working Copy/Paste Engine) transfers the data from the temporary section to the domain area. During this operation the pasting algorithm distributes the resulting Nodes by their corresponding Partitions. The consistency of the underlying references is guaranteed at this stage.

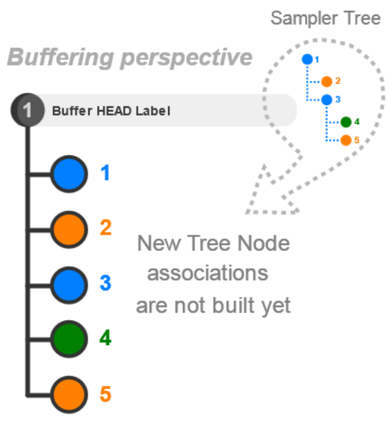
4.3 Copy/Paste Workflow

The Copy/Paste Tool relies on Copy/Paste Engine: the auxiliary tool which holds the actual data transferring algorithm. It provides methods to copy & restore the user tree structure to & from the internal buffer. Such copying must be smart enough in order to preserve the Tree Node organization of the source objects (this is not carried out by standard OCAF services) and adjust the transferred references according to the requirements described above (this is quite specific and obviously not covered by native OCAF services).

The internal workflow of the Copy/Paste Engine can be broken down onto several consequent steps described below.

4.3.1 Flattening

Given the source user-tree of Data Nodes being copied, the Engine iterates over this tree performing deep copying of each item. This process is referred to as flattening as Copy/Paste Engine places the prepared copies one-by-one into the buffering section (Copy) or corresponding Partitions (Paste). Node types are mixed up in the buffering section: no sub-Partitions are created here for simplicity.



Once this stage is completed, we have copy/paste buffering section filled with the well-formed copies of the initial Data Nodes. However, these Nodes do not have any Tree Node connectivity yet (as `TDataStd_TreeNode` attributes are intentionally filtered out during the copying process) and do have *inconsistent* reference pointers. These issues are resolved by the following steps of transferring routine.

4.3.2 Sampler Tree

The Tree Node connectivity is rebuilt in a consistent form by means of a dedicated *Sampler Tree* object representing a tree of source Node IDs. This tree plays a role of recovering sampler as it completely repeats the topological structure of the initial source Node tree. This tree is prepared at the previous step along with `Relocation Table`: a specific data map storing the correspondence between source labels and the new copied ones. Sampler Tree is iterated against the Relocation Table in order to build the resulting Tree Node connectivity.

Note

Actually, there is no big necessity to introduce such Sampler Tree as we can use the initial Node connectivity for the purpose of topological rebinding. However, Sampler Tree seems to be a good lightweight abstraction which can also be used as an input to other Data Model algorithms dealing with custom trees of Data Nodes. The benefits from using such a notion instead of the source data directly are as follows: it remains immutable during the copy/paste procedure, while we cannot guarantee that for the initial topology; it represents pure topological connectivity without any binding to domain-specific data unlike `TDataStd_TreeNode` objects which are essentially OCAF attributes bound to raw OCAF labels.

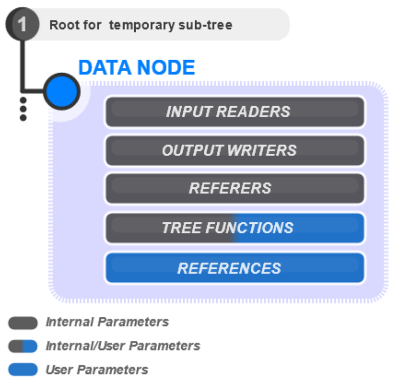
After the second step completes, we have a well-formed topological structure of the cloned Data Nodes. The only remaining inconsistency here is the presence of invalid references (if any such references exist).

4.3.3 Normalization

Each Data Node can contain several types of references, excluding Tree Node Parameters which were discussed above. These reference types are listed below:

- DIRECT references:
  - Tree Function Parameters: they have references to their INPUT and OUTPUT arguments;
  - Plain Reference Parameters: they have explicit reference to an arbitrary Node or Parameter;

- BACK References:
  - INPUT Reader observers: references to those Tree Function Parameters which read some data from the current Node;
  - OUTPUT Writer observers: references to those Tree Function Parameters which write some data to the current Node;
  - REFERER observers: references to those Reference Parameters which point to the current Node.



The common rule is to classify all references against the Sampler Tree onto two categories: *in-scoped* references and *out-scoped* references. The reference is said to be in-scoped if it does not point outside the Sampler Tree anyhow. The reference is said to by out-scoped if it has at least one pointer out of the Sampler Tree. The in-scoped references are adjusted regarding to the Relocation Table and pushed to the resulting copy. The out-scoped references are thrown away or kept as-is, depending on the user's choice. The default behaviour consists in ignoring all out-scoped references except the references to Variables. However, the client code is free to charge *Reference Filter*, enumerating those references which have to be transferred to the destination sub-tree even being in out-scoped state.

In fact, custom Data Model class must set its custom reference filtering rules by implementing a dedicated pure virtual method:

```
//! Populates the passed collections of references to pass out-scope filtering
//! in Copy/Paste operation.
//! \param FuncGUIDs [in/out] Function GUIDs to pass.
//! \param Refs [in/out] Reference Parameters to pass.
void MyDemo_Model::invariantCopyRefs(ActAPI_FuncGUIDStream& FuncGUIDs,
                                     ActAPI_ParameterLocatorStream& Refs) const
{
  FuncGUIDs << MyDemo_MeshFunc::GUID();

  Refs << ActAPI_ParameterLocator( STANDARD_TYPE(MyDemo_AnimationNode)->Name(),
                                   MyDemo_AnimationNode::PID_TopoReference )
       << ActAPI_ParameterLocator( STANDARD_TYPE(MyDemo_MeshNode)->Name(),
                                   MyDemo_MeshNode::PID_TopoReference );
}
```

The mentioned `invariantCopyRefs()` is called automatically by the framework each time the copy/paste operation is launched. This method enumerates those references which must pass the out-scope filtering. The example above illustrates the case when such references are defined once and for all. However, you are free to implement this method in a way providing dynamic changing of filter invariants. Notice that normally the passed collections are already charged with some framework-specific references (e.g. Variable references), so make sure not to clean up their contents (unless you clearly understand what you are doing).

Note

> In case of paste workflow, we can obtain so called out-scoped "ghost" references. These are the references corresponding to non-existent data. E.g. if a Data Node was connected to a Variable which was deleted between copy & paste invocations, the buffered representation of such Data Node will contain a reference to the "dead" Variable. Out-scoped "ghost" references are always removed, regardless of any user-defined filtering as it is essentially a question of data consistency.

The separation between in-scoped and out-scoped references is performed by means of *Reference Classifier* tool provided along with Copy/Paste Engine tool. Given a reference, it checks whether this reference is enumerated in the Sampler Tree or not. If so, the reference is classified as in-scoped, otherwise — as out-scoped. The unified abstract for any kind of reference is a simple OCAF label which plays as an atomic pointer to data in such standard OCAF attributes as `TDF_Reference` and `TDataStd_ReferenceList` — the only kinds of reference holders used in the Active Data framework.

Note

> A special care is required for those Parameters which belong to the copied Node and are referenced by this Node as well. This case mostly concerns Tree Functions which are binding several input Parameters of the owning Node to some output Parameters of the same Node. This case is specific only due to OCAF internals as OCAF has its own vision on treating such in-house references. In terms of OCAF, the label is said to be "self-contained" if all its references are local to this label with its storage-tree children (there is no word about Tree Node connectivity here). Such local references are exchanged automatically by OCAF, introducing some elementary relocation logic on a portion of existing references. This automation seems to be a good thing on the plain data hierarchies, but it brings additional complexity into our in-scoped/out-scoped filtering as the corresponding classifier must be aware of such "helpful" intentions from OCAF side. See unit-test case 3.5 for illustration.

4.4 Unit tests

Copy/Paste Engine tool is tested by means of `ActTest_CopyPasteEngine` class. Here we describe the available tests.

4.4.1 Case 1

Test if plain Data Node is copied without any existing user-tree hierarchies and references into the target Node containing no other children. The prepared copy should be completely independent from the source data. The latter is checked by affecting the source data between copy and paste invocations.

4.4.2 Case 2

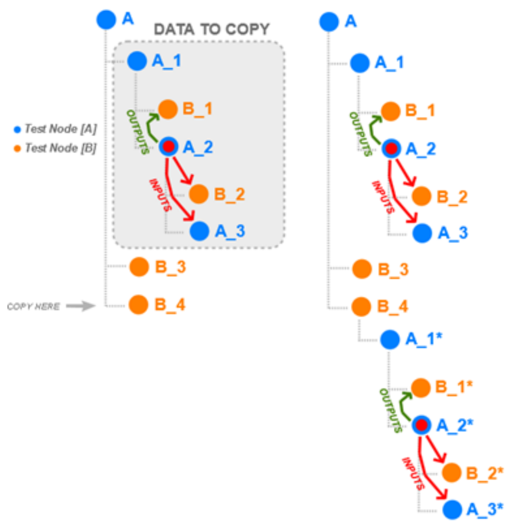
Test if the source user-defined sub-tree is completely copied to the target Data Node containing no other children. The source connectivity of Data Nodes should not be affected. The copied hierarchy must be identical to the source one. Consistency of references is not checked at this stage.

4.4.3 Case 3

Test if plain Data Node with Tree Function references established as shown on the following schemes is copied properly to the target Data Node.

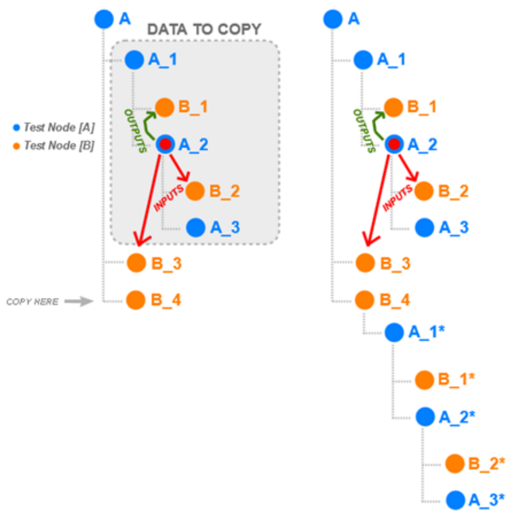
4.4.3.1 Case 3.1

Only in-scoped references exist. Nothing should be lost. Tree Function should pass to the copy with adjusted Parameter references. INPUT Readers & OUTPUT Writers for the involved Data Nodes must also be adjusted correspondingly.



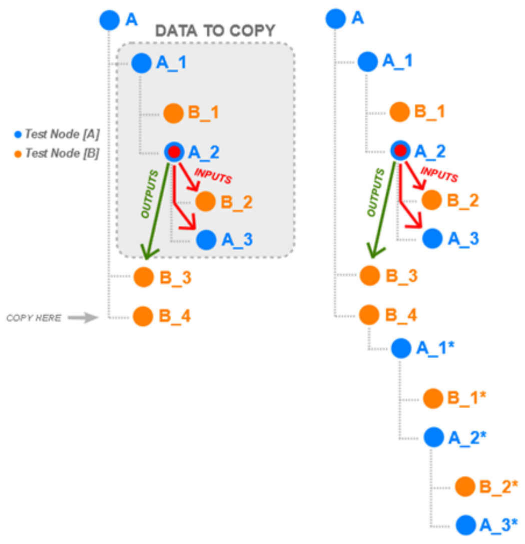
4.4.3.2 Case 3.2

Out-scoped references exist in INPUT set of Tree Function Parameters. Tree Function must be disconnected. The corresponding back-references (pointing to the Tree Function Parameter being disconnected) in Nodes B_1 and B_2 must be removed. Data Node B_3 must not be affected anyhow.



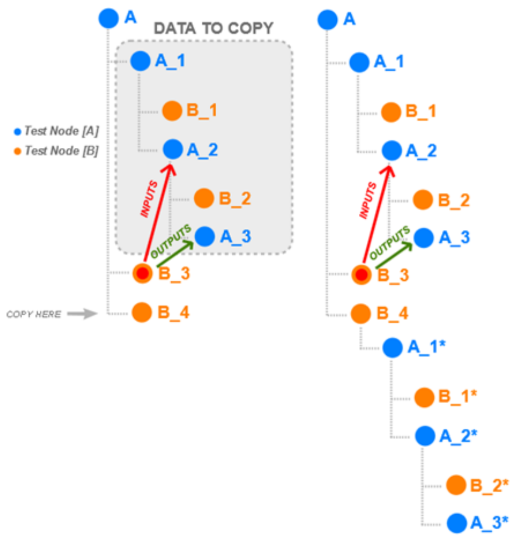
4.4.3.3 Case 3.3

Out-scoped references exist in OUTPUT set of Tree Function Parameters. Tree Function must be disconnected. The corresponding back references must be cleaned up from Data Node B_2 and A_3. Data Node B_3 must not be affected anyhow.



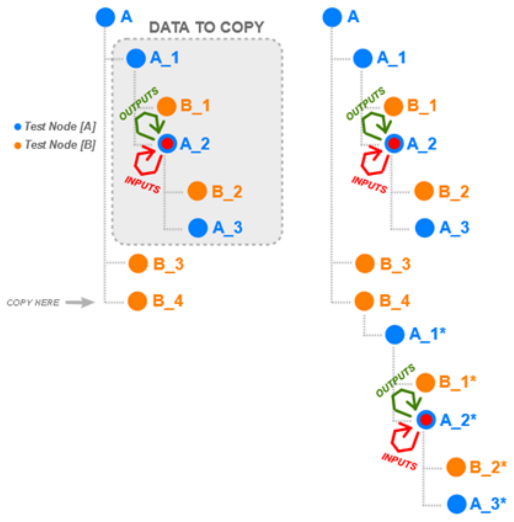
4.4.3.4 Case 3.4

Tree Function is fully external to the sub-tree being copied. All the corresponding back-references (INPUT Readers and OUTPUT Writers) must be cleaned up from the copies of Data Nodes A_2 and A_3.



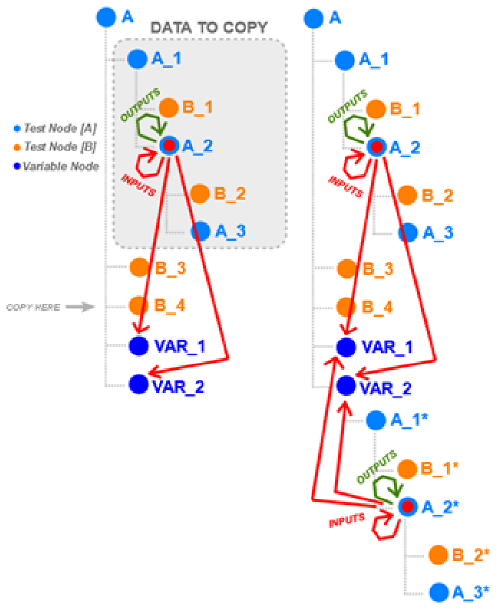
4.4.3.5 Case 3.5

Tree Function is fully local and corresponds to those called self-contained label in OCAF terms. In this case OCAF partially participates in the relocation logic, so our Scope Classifier must understand such peculiarities and mark the Tree Function as in-scoped.



4.4.4 Case 4

Test on evaluation Tree Functions: they must not be lost.

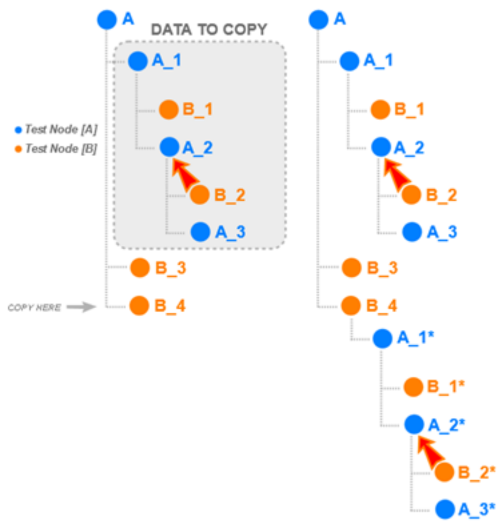


4.4.5 Case 5

Tests on Reference Parameter.

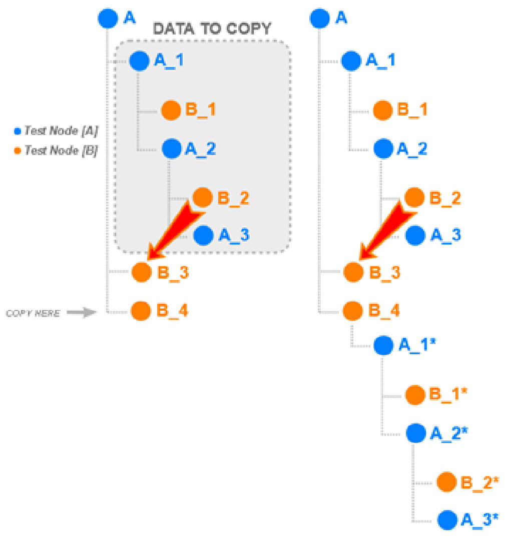
4.4.5.1 Case 5.1

In-scoped test. Reference must be converted correspondingly.



4.4.5.2 Case 5.2

Out-scoped test. Reference must be disconnected.



4.4.5.3 Case 5.3

Out-scoped test with exceptional reference filtering. Reference must be preserved as-is.

4.4.5.4 Case 5.4

Out-scoped test with exceptional reference filtering. The copy/paste operation is performed twice: first time for the source Node and second time for its copy. Reference must be preserved as-is in both cases, demonstrating that Reference Filter propagates its rules on the newly appeared Nodes as well.

4.4.5.5 Case 5.5

Out-scoped test with exceptional reference filtering. Check if the reference filtering works on the opened document as well (it is a transient information, not stored in OCAF, so it should be re-populated when Data Model is opened. At least we can re-populate it on each Copy request).

4.4.6 Case 6

Test if our custom Mesh Attribute is properly copied.

4.4.7 Case 7

Test on pasting Data Node with out-scoped "ghost" reference in the Tree Function arguments list. The corresponding Tree Function must be disconnected.

4.4.8 Case 8

Test on pasting Data Node with out-scoped "ghost" reference in Tree Function results list. The corresponding Tree Function must be disconnected.

4.4.9 Case 9

Test on pasting Data Node with out-scoped "ghost" reference in Reference Parameter. The corresponding Reference must be cleaned up.

Chapter 5

Mesh

Todo This section will describe custom Mesh Attribute available in Active Data framework. Currently this description is not available.

Chapter 6

Undo & Redo

6.1 Overview

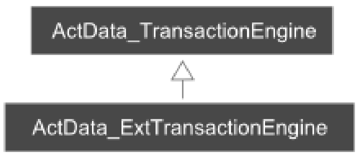
Undo/Redo (shortly UR) functionality shipped with Active Data framework is fully implemented at the level of Data Model class. Basically, the Undo/Redo mechanism fully benefits from the standard OCAF UR functionality which is an important part of the "rapid application development" tool set. While this default OCAF UR engine makes a significant deal of stacking the so called *Modification Deltas*, it is not high-level enough to be used without some additional amelioration for Active Data models. The detailed requirements to Undo/Redo functionality in Active Data are described in the following section.

6.2 Requirements

Active Data framework must provide data-level Undo/Redo functionality basing on the following requirements:

- As AD framework comes with Tree Function propagation mechanism expressing the concept of active data model, the Undo/Redo functionality must organically fit this concept. That is, given a valid state of a data model, UR functionality must result in another (expected) valid state. We can imagine three typical scenarios of Active Data framework usage from Tree Function connectivity perspective:
  - Tree Function mechanism is not utilized. If AD framework is used as a foundation for passive data storage without any algorithmic connections, Undo/Redo might not care about functional dependencies at all;
  - Tree Function dependency graph is executed in the same transaction that affects its initiating data. In that case the Tree Functions being executed are just the mean of data model modification which works in a "sand box" of transactional scope. Undo/Redo functionality is not aware of whether some Tree Functions were actually executed or not, so there are no differences comparing to the previous case;
  - Tree Function dependency graph is executed in a transaction different from the one affecting the initiating data. The peculiarity here is that UR functionality must be able to recover the LogBook records when switching between the data model states. This point makes the usage of OCCT standard transient (here it means "not bound to OCAF document") LogBook implementation ineligible as doing so we will get no easy way to provide such recovering. That is why OCAF default transient `TFunction_Logbook` class cannot be used by Active Data Framework. Instead, it has to come with its own LogBook object attached to the OCAF document in a usual persistent way. Thereby, the records in this dedicated OCAF section will also be covered by UR functionality.
- It should be possible to shoot several Undo/Redo operations at once. For simplicity, this functionality can be implemented by sequential applying of all intermediate Modification Deltas preceding the ultimate one inclusively;
- There should be possibility to bind application-specific data to each transactional scope (i.e. Undo/Redo Modification Delta). E.g. for the sake of better GUI ergonomics, each Undo/Redo grain can have an associated

Unicode name describing what is going on. However the possibility to have anonymous transactions should be kept as well. This diversity of design choices can be easily expressed by introducing different kinds of *Transaction Engines* — service providers for deployment and management of transactional scopes. That is, one type of Transaction Engine will be responsible for the anonymous transactions, and another — for those having some application-specific data, such as name (but not limited to it). The desirable Engine type must be chosen by application developer at design time. Data Model instance handles only one immutable Transaction Engine (possibility to change Transaction Engine instance dynamically seems to be not only useless, but also error-prone);



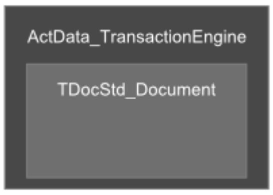
- The following specific operations must not be the subjects of UR:
  - Pre-population of Data Model on `NewEmpty()` invocation;
  - Compatibility conversion.

6.3 Transaction Engine

Architecturally, Transaction Engine is a Data Model co-worker tool providing a set of services for deployment and management of OCAF transactional scopes. Technically, Transaction Engine is a wrapper under the standard OCAF transaction mechanism plus additional custom services, e.g. binding of application-specific data to transactional scopes.

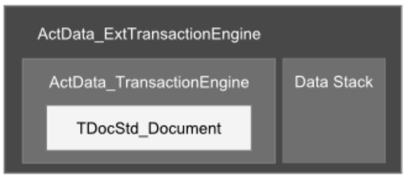
6.3.1 Simple Transaction Engine

Simple Transaction Engine declares a comprehensive interface for managing the transactional scopes. It works as a pure wrapper for OCAF native `TDocStd_Document` transactional functionality delegating the most of operational requests to the standard OCCT services. It is also a base class for other customized Transaction Managers. If you do not need any customization (such as transactional naming), use Simple Transaction Engine.



6.3.2 Extended Transaction Engine

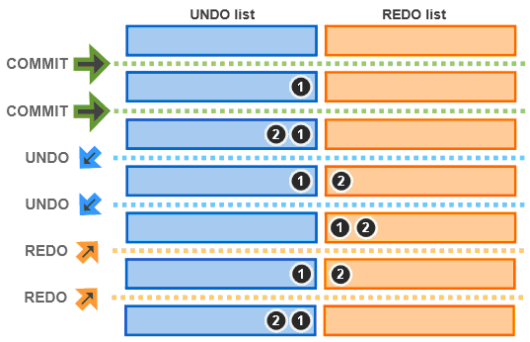
Besides the usage of standard OCCT transaction services, Extended Transaction Engine introduces a data binding extension to the managed Modification Deltas. It requires that any transaction being committed is bound to some user-specific data structure even if such structure is empty. As OCCT OCAF component does not support such binding functionality, Extended Transaction Engine comes with its own collection of Undo/Redo records storing the mentioned data structures for the correspondent Modification Deltas. It is only an extension, not re-inventing the wheel, as the standard UR stack is still utilized by OCAF internally.



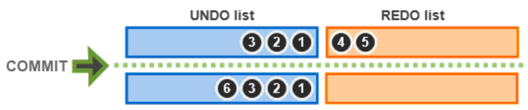
Note

> Actually, the Extended Transaction Engine is a variation of a concept introduced by OCCT Multi Transaction Manager shipped in `TDocStd` package. The latter tool is used for supporting named transactions in multi-document environment. We do not re-use this component due to its excessive complexity for Active Data framework needs and because it is limited to working with names (while we want to bind any kind of data).

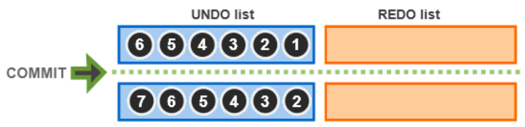
Internally Data Stack is implemented as two plain ordered collections storing the instances of `TxData` structures. `TxData` structure in-turn allows storing a dynamically growing ordered collection of primitive types, such as integer, real, Boolean and string. `TxData` structures controlled by Extended Transaction Manager are re-distributed between Undo and Redo collections depending on the current state of data modification session represented by several latest transactions. Such re-distribution is a full reflection of the processes ongoing in OCAF UR mechanism with Modification Deltas internally.



New item appears in Undo collection once user commits a transaction. `TxData` items are accumulated there and do not pass to Redo collection until user calls Undo method. If user commits another transaction while the Redo collection is non-empty, the Redo collection is cleaned up. This practice is commonly accepted and established in OCCT OCAF UR mechanism.



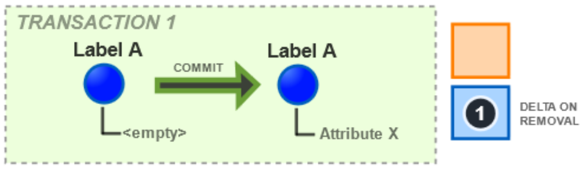
In order to prevent Undo/Redo stack from consumption of too much memory, OCAF introduces Undo Limit integer value which can be optionally specified for each working OCAF document. If the size of Undo collection exceeds the specified Undo Limit, the last stored Modification Delta (representing the oldest change) is thrown away from Undo collection. This is automatically done by OCAF in its default `CommitTransaction()` method of `TDocStd_Document` class. Extended Transaction Engine comes with a reflection of this Undo Limit logic for the managed `TxData` items. Undo Limit value utilized by Active Data Transaction Engines is the same as for OCAF Document.



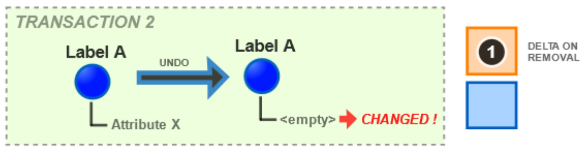
6.4 Working out of transaction

Normally you are highly advised never to change your data out of transactional scope. However, sometimes you do not want your modifications to appear in Undo/Redo stack or you may have different reasons to avoid transaction opening (e.g. not to back-up your attributes). In such cases it is possible to disable transactions. However, this operational mode should be used with a great care. To demonstrate that mixing transactional and non-transactional approaches is not a safe way to manipulate your data, we take a simple example.

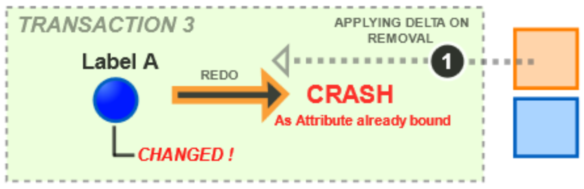
Let label A get attribute X in transaction 1. A standard Delta On Removal is produced by OCAF and pushed to Undo stack:



Delta On Removal is a specific object which drops the newly created attribute X once Undo command is invoked. If user calls Undo, Delta On Removal is relocated to Redo stack and label A looses its attribute X. Let us assume that we inject some non-transactional logic at this stage and associate some attribute Y to label A once Undo is completed.



Everything works well until we call Redo command. It takes Delta On Removal from Redo stack and attempts to apply it on our "dirty-changed" label A. The Delta On Removal is used by OCAF with a "negative sign" on Redoing, so OCAF engine attempts to create attribute X again on the label A. However, our non-transactional intervention has already bound the attribute Y to label A, and new attribute cannot be created so. Indeed, it is forbidden to set an attribute to OCAF label if the attribute of the given type already exists (we assume that attribute Y is of the same type as attribute X, e.g. integer). OCAF fails.



This is an example of inaccurate mixing of transactional and non-transactional approaches. If you still think that you want to use this trick, here is the bunch of methods for disabling/enabling transactional scope:

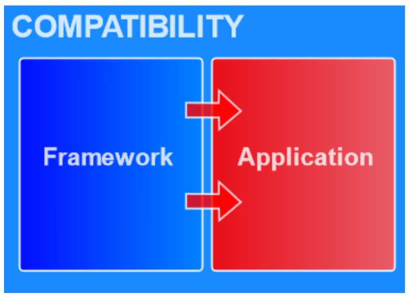
```
ActData_TransactionEngine::DisableTransactions();
ActData_TransactionEngine::EnableTransactions();
```

Chapter 7

Backward Compatibility

7.1 Overview

Backward compatibility conversion mechanism for applications based on Active Data framework is conceptually composed of two components: framework's data converter and application's data converter.

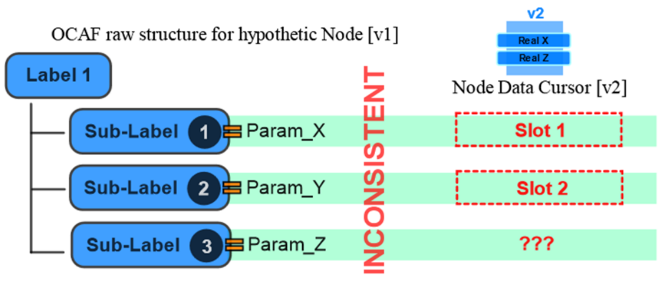


This dualism is natural as it comes from the fact that versions of AD framework and custom application may evolve independently. E.g. a newer version of Active Data can re-organize the way how actual data chunks are stored in OCAF, but it cannot influence the structure of the actual domain-specific data model grounded on the framework. On the other hand, custom application is free to change the contents of its particular data model, but it should not care of the changes in the storage scheme which can appear after porting to a newer version of the framework. The first kind of evolutionary conversion must be performed automatically by the framework, while the second one remains under responsibility of the application developer.

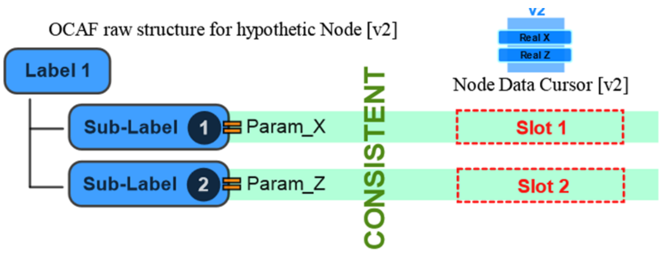
The following illustration highlights the necessity of the mentioned conversion process appealing to the technical organization of data. Let us consider that we have a Data Node composed of three Parameters: X, Y and Z. These Parameters could represent some point in 3D space. Internally, in order to have such Node in your application, you derive your Node class from `ActData_BaseNode` class and declare enumeration for your Parameter IDs:

```
enum ParamId
{
  PID_X = ActData_BaseNode::UserParam_Last,
  PID_Y,
  PID_Z,
  PID_Last = ActData_BaseNode::UserParam_Last + ActData_BaseNode::RESERVED_PARAM_RANGE
};
```

Now let us check what will happen if we decide to remove Parameter Y in the succeeding version of the application. Without having the corresponding compatibility conversion performed, your actual Data Cursors (Nodes in that case) will become inconsistent with the loaded OCAF raw data. Indeed, new Data Node (with `PID_X` and `PID_Z` Parameters only) will be charged with more comprehensive raw data which is still containing Parameter Y. Actually, being settled down onto non-adjusted OCAF source, your recent Data Node will connect the persistent Y value to its `PID_Z` slot, while the persistent Z value itself will remain uncharged. The problem is even more insidious as such Data Node will be WELL-FORMED from the framework's point of view. Indeed, being unsynchronized with primary OCAF data source, it is nevertheless loaded will all necessary data of expected types (Y and Z are both Real Parameters in our case).



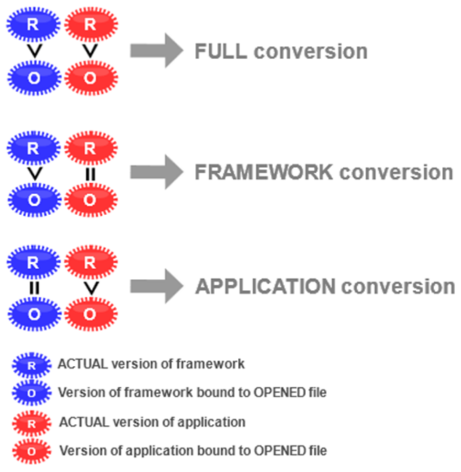
In order to fix that problem, we need to remove OCAF label number 2 from the imported CAF source and re-number (re-tag) label 3 to have ID equal to 2. This simple case is fully domain-specific, so the conversion routine performing such "fix" must be implemented at application level.



To make separation of framework/application conversion routines feasible, two version identifiers are bound to any data model grounded on Active Data framework:

- Version of the Active Data framework which evolves along with the framework;
- Version of the application which evolves along with a particular application based on AD framework.

The backward compatibility conversion process is launched automatically depending on the relations between the actual version identifiers and those restored from the project file being opened.


R > O, R > O FULL conversion
R > O, R = O FRAMEWORK conversion
R = O, R > O APPLICATION conversion
R ACTUAL version of framework
O Version of framework bound to OPENED file
R ACTUAL version of application
O Version of application bound to OPENED file

If any actual version identifier is less than the version identifier coming from the opened project file, the compatibility conversion is impossible. This situation is treated as attempting to open a newer version of data model in the older environment.

In order to enable compatibility conversion mechanism in your application, it is necessary to use dedicated toolkit shipped with the Active Data Framework. This section describes the entire conversion process and gives a list of available tools.

7.2 CAF Converter

The evolutionary conversion of data model structures is approached by CAF Converter tool represented by `ActData_CAFConverter` class in `Tools` sub-package of `ActiveData` library. The instance of this class is charged with a set of static functions performing the actual upgrade logic. The oldest supported framework's version is 1.0.0 (this is the first release).

CAF Converter tool must be enriched with a dedicated conversion method each time the version of the framework or application is incremented. All conversions are performed consequently, i.e. upgrading from hypothetical version 1 to version 3 will be carried out as conversion for version 1 to version 2 and then from version 2 to version 3. Thus the framework developer should take care of modification deltas between the consequent pairs of framework versions only.

As a reflection of data conversion dualism, `ActData_BaseModel` class utilizes two CAF Converter instances: one for the framework conversion and another for the application conversion. As framework conversion is carried out automatically, the corresponding instance of CAF Converter is allocated and initialized by the framework itself. On the other hand, in order to enable the application-specific conversion mechanism, the application developer should implement pure virtual `converterApp()` method returning the properly charged CAF Converter instance. E.g:

```
Handle(ActData_CAFConverter) MyDemo_Model::converterApp()
{
  return new ActData_CAFConverter(
    ActData_ConversionStream() << ActData_ConversionTuple(VersionLog_Lot1Iteration4,
                                                         VersionLog_Lot2Iteration1,
                                                         APP_CVRS::v040_to_v050)
                               << ActData_ConversionTuple(VersionLog_Lot2Iteration1,
                                                         VersionLog_Lot2Iteration2,
                                                         APP_CVRS::v050_to_v060) );
}
```

If you are not planning to have any compatibility conversion mechanism in your application, just return a NULL value from the mentioned method. It will prevent framework from opening the older versions of OCAF documents and lead to graceful releasing of internal resources allocated during the loading operation.

Notice that due to incremental nature of the conversion process, you should provide continuous set of version deltas along with their correspondent conversion routines. You are not forced to respect the versions order here. However, you should not introduce version gaps when charging your domain-specific CAF Converter. E.g. the following initialization code is faulty:

```
Handle(ActData_CAFConverter) MyDemo_Model::converterApp()
{
  return new ActData_CAFConverter(
    ActData_ConversionStream() << ActData_ConversionTuple(VersionLog_v1,
                                                         VersionLog_v2,
                                                         APP_CVRS::v1_to_v2)
                               << ActData_ConversionTuple(VersionLog_v3,
                                                         VersionLog_v4,
                                                         APP_CVRS::v3_to_v4) );
}
```

Indeed, being initialized in such a way, CAF Converter cannot perform incremental upgrade as there is a gap between version deltas: conversion from v2 to v3 is not specified. The following correction will make it work:

```
Handle(ActData_CAFConverter) MyDemo_Model::converterApp()
{
  return new ActData_CAFConverter(
    ActData_ConversionStream() << ActData_ConversionTuple(VersionLog_v1,
                                                         VersionLog_v2,
                                        APP_CVRS::v1_to_v2)
                               << ActData_ConversionTuple(VersionLog_v2,
                                                          VersionLog_v3,
                                                          APP_CVRS::v2_to_v3)
                               << ActData_ConversionTuple(VersionLog_v3,
                                                          VersionLog_v4,
                                                          APP_CVRS::v3_to_v4) );
}
```

Notice that the mentioned routines like v1_to_v2 are fully under responsibility of the application developer. It is a good practice to package all such routines into a dedicated namespace (`APP_CVRS` here) or even library.

Even though the application-specific conversion process might perform any juggling with data and cannot be formalized so, it is nevertheless most often composed of series of atomic data modification requests. E.g. such typical operations as removing a Parameter from a Data Node or adding a new Parameter to the given place are not domain-specific, while their composition is. That is why Active Data framework comes with a tool set implementing all standard upgrading routines. This tool set contains the following utilities:

- CAF Conversion Asset
- CAF Conversion Context

Both tools are described below.

7.3 CAF Conversion Asset

CAF Conversion Asset allows you performing some well-defined operations on your data model being converted. This is actually a container for useful conversion logic which can ease your domain-specific conversion. The collection of ready-to-use operations can grow with new releases of Active Data framework. Here we give an illustrative extraction of some useful functions which are currently available in Conversion Asset:

- ActualizeVersions: updates the source versions (application plus framework) of OCAF document to the actual ones. This is the most obvious and primary upgrading routine which have to be performed in any case, even if there is no actual need to perform domain-specific conversion of user data;
- ChangeParameterType (obsolete): changes the type of the given (existing) Parameter to the requested one. This method will clean up the initial Parameter data without a care of any references which might exist in other Data Nodes or even in the same Node. Ergo, use this method with care and do not forget to update the referrers if any. Returns the new Parameter instance of the desired type. This instance is a BAD-FORMED Data Cursor ATTACHED to the CAF source and ready to be populated with actual data so.

7.4 CAF Conversion Context

You could notice that Parameter removal and insertion functionality is not provided by Conversion Asset. Generally, these modifications cannot be easily achieved at a single pass. For instance, it is not enough to delete a Parameter data from the Node's OCAF source as the corresponding OCAF label might have been referenced by other Nodes somehow (via Tree Function, Reference Parameter etc). In this case all such references have to be normalized. Moreover, there is a known OCAF limitation concerning deletion operation. It is not possible to remove any label physically from transient OCAF structure. While the latter limitation makes a good deal for Undo/Redo mechanism (we always have some label to apply modification delta on), it significantly complicates any elegant approach to data conversion.

CAF Conversion Context is another tool in framework's conversion gentlemen set. It addresses the following issues:

- Insertion/Removal/Modification of Nodal Parameters. Whenever application developer wants to perform such elementary operation, he needs to identify the target Parameters in an exact way. Conversion Context presumes that application developer *always works with original local Parameter IDs* (PIDs). E.g. you can ask

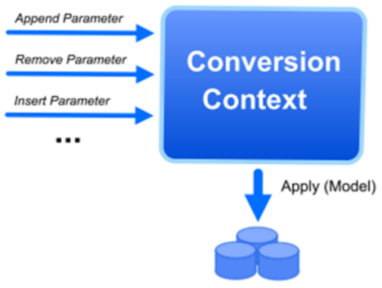
Conversion Context to insert something before "this original PID". It works regardless of whether the referenced PID corresponds to Parameter being removed or not. While such an approach to addressing Parameters is logically transparent, it, however, requires from application developer to know what PIDs have been used in the previous versions of the application;

- Normalization of references. Conversion Context not only applies atomic modifications, but also ensures data consistency in terms of established logical connectivity between Nodal Parameters. Tree Node connectivity is also taken into account, ensuring that parent-child relations between data objects are not lost (no special efforts are needed here as Tree Nodes are established in internal section of Nodal Parameters which are out of scope of compatibility conversion);
- Fighting "ghost" labels. Conversion Context takes advantage of OCCT persistence filtering rules which throw out any dummy non-referenced labels once OCAF document is saved and restored. Therefore, the mentioned problem with dummy labels remaining after deletion is resolved via sequential saving and restoring OCAF document using a temporary file.

7.4.1 General description

As mentioned above, in order to assure that atomic modifications on the data model are performed in a consistent way, one should utilize Conversion Context tool rather than invoke low-level conversion utilities manually. Conversion Context works like a record book accumulating all modification requests to be applied on the data model. Once all such requests are gathered, user invokes method `Apply()` in order to get them propagated to the model physically.

The philosophy behind this Conversion Context tool is the same as in Re-Shaper utility provided by OCCT geometric kernel for topological shapes. At the first stage all modification requests are recorded, but no one is applied. Moreover, at this stage we do not care of the actual data model instance being converted, so all modification records are fully abstract. At the second stage the modification requests are applied on a particular instance of a data model provided by user. Here Conversion Context ensures that their execution scheme is consistent and efficient.



7.4.2 Approach to unit testing

Unit tests are performed on test data model instances of different structures. Each particular type of data model has strictly defined contents and connectivity between its underlying Nodes. The following specific types of test models are supported at the present day:

- ABC-01

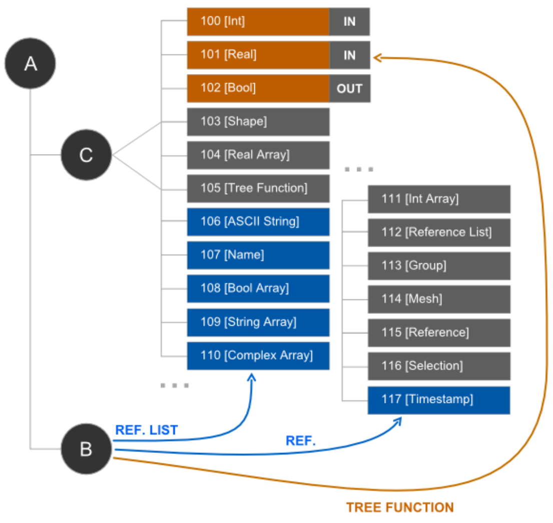
All tests concern only Conversion Context tool and organized in the following way:

- Sample data model is created;
- Instance of Conversion Context tool is used to perform different modifications on this test model;
- Once conversion is completed, the resulting project is dumped into file using CAF Dumper functionality (the most verbose dumping mode is used here);

- The output file is compared with referential one. The validity of the referential file is ensured by its preliminary manual calibration. Such calibration is performed only once (when unit tests are implemented).

Atomic conversions are applied to a dedicated sample Data Node containing Parameters of almost all supported types, except those which are designed for internal usage only (Tree Node Parameter, Type Name Parameter etc).

The general structure of test project ABC-01 is illustrated on the following drawing:



Test project ABC-01 is organized in both simple and illustrative way. Root Node A contains two children of different types: B and C. The Node of type C is a working Node which will be affected by Conversion Context tool. The Node of type B plays a role of external referrer. It holds three types of references to Node C:

- Reference List Parameter;
- Reference Parameter;
- Tree Function Parameter (it has two inputs and one output).

7.4.3 Conversion requests

Conversion Context supports the following main atomic operations:

- Insertion of a Parameter, including special treatment for appending case;
- Modification of a Parameter;
- Removal of a Parameter.

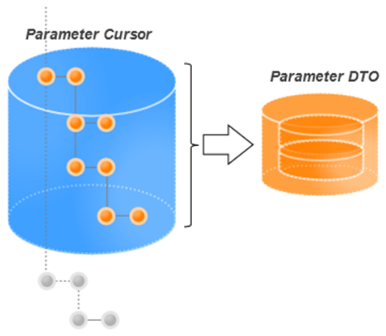
One could easily notice that this triple is similar to well-known Create/Update/Delete data modification operations referred to by CRUD abbreviation ("R" stands for Read and is out of our interest here). These operations form a basis for any, even sophisticated, data morphing routine. The mentioned atomic operations can be mentally classified into the following groups:

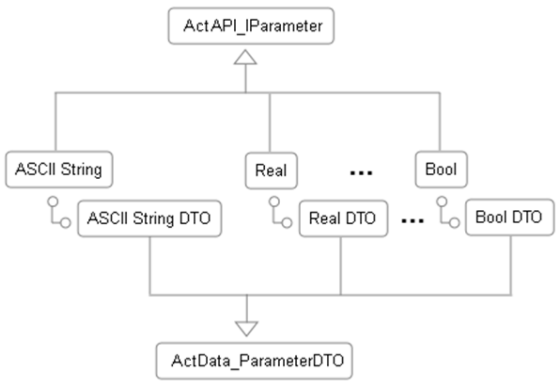
- Reference-safe;
- Reference-unsafe.

Reference-safe operations do not affect logical relationship between data objects. On the contrast, reference-unsafe operations do. The only reference-safe operation is updating as it prohibits any crucial data modifications.

The remaining operations are reference-unsafe and require additional normalization stage to be performed on entire set of data model references.

Even though all atomic operations concern Nodal Parameters, they do not actually work with Parameter instances. Indeed, Parameter is a transient Data Cursor which cannot be used for transferring data without being settled down onto OCAF structures. That is why we use a concept of DTO (Data Transfer Object) to represent Parameters from the conversion point of view. DTO is a well-known design pattern used to transfer data between service and persistence architectural layers.





DTO is composed of global Parameter's ID (GID) and the actual data being transferred. GID is immanent to any D-TO type. It contains local Parameter's ID (PID from application-specific enumeration) and Node's ID which together allow referring to any Parameter in a unique way.

The following rules are established for modification recording:

- One Parameter GID can correspond to a single type of modification only. I.e. it is not possible to record Modification and Removal, Removal and Insertion or Modification and Insertion at the same time for the given GID. In order to do such things, you have to use several Conversion Contexts consequently. Still, it is possible to record insertion "before this ID" multiple times as this operation is logically transparent (new Parameters will appear in the same order they were recorded);
- Conversion Context itself is responsible for keeping Parameter tags in a contiguous form. This requirement reflects the most intuitive way of Parameter tagging in scope of their Data Nodes. It means that normally application developer just adds or removes items from the corresponding enumeration containing all Parameter IDs:

  Version 1 (v1):

  ```
  enum ParamId
  {
    PID_X, // 100
    PID_Y, // 101
    PID_Z  // 102
  };
  ```

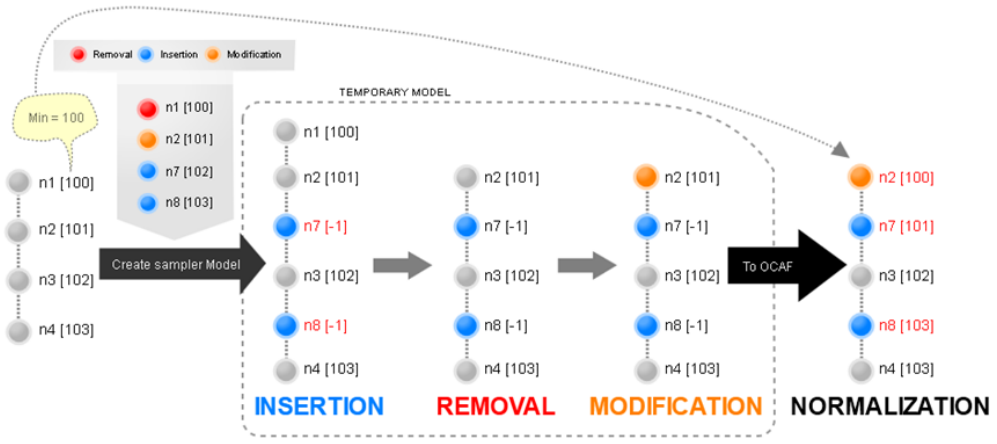
Version 2 (v2):

```
enum ParamId
{
  PID_X, // 100
  PID_Y, // 101
  PID_A, // 102
  PID_B  // 103
};
```

If application developer uses custom tagging scheme, compatibility conversion should not utilize Conversion Context. In such exotic cases application developer should take care of compatibility conversion process manually.

- As already mentioned above, Insert/Update/Remove operations are specified with use of original Parameter IDs.

The entire workflow automated by Conversion Context is illustrated by example below:



In this example we convert sample Node composed of four Parameters with tags in range 100-103. Notice that according to the rules mentioned above, Parameter tags should be contiguous.

Conversion Context is initialized with four modification requests (remove, update and two inserts). Notice that each modification request is followed by basis PID, which is essentially an original PID for data being converted. More precisely, the meaning of the basis PID is the following:

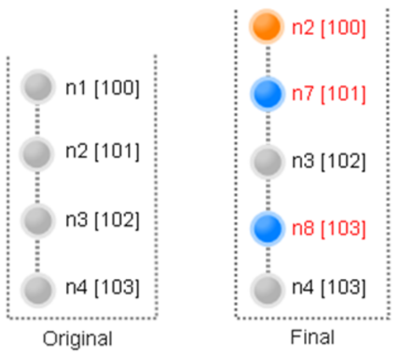
- Deletion. Original PID of Parameter to remove;
- Modification. Original PID of Parameter to update;
- Insertion. Original PID of Parameter to insert new one before.

Insertion logic presumes that new Parameter being added to the data model will get a new PID automatically. Application developer does not care of its PID at all. Application developer rather grounds his conversion logic on the knowledge of original PID tags in a Data Node being converted.

7.4.3.1 Temporary Conversion Model

Conversion Context uses a notion of temporary Conversion Model in order to represent the entire data morphing process. In fact, theoretically it is also possible to apply modification requests directly on the target OCAF data structures. Such direct approach, however, involves a necessity to perform granular modifications in OCAF tree, including re-tagging of labels, insertion of labels, partial removal etc. OCAF is not designed well for such kind of operational cases, so Conversion Context proceeds in a different way.

All changes introduced during conversion deployment are recorded in a dedicated storage called Conversion Model. This storage has nothing to do with OCAF, and so it is free from the mentioned limitations.



Conversion Model distributes its data by Conversion Nodes. Each Conversion Node contains Conversion Parameter objects corresponding to a particular Node involved in compatibility conversion. Conversion Parameter is a simple wrapper under Parameter DTO enriched with conversion history information. History includes evolution mark (None, Updated, New, Deleted) and original PID for the Parameter (if it is not a NEW one).

Once conversion is performed in a transient way for Conversion Nodes, their contents are dumped back to OCAF tree at once. In order to do this, the corresponding OCAF Nodal branches are cleaned up and completely re-populated from Conversion Model.

7.4.3.2 Insertion and appending

Insertion modification request is useful if the newer version of environment re-organizes a set of reserved Parameter IDs, introducing new ones at desired positions.

Version 1 (v1):

```
enum ParamId
{
  PID_X, // 100
  PID_Y, // 101
  PID_Z  // 102
};
```

Version 2 (v2):

```
enum ParamId
{
  PID_X,  // 100
  PID_X1, // 101
  PID_Y,  // 102
  PID_Y1, // 103
  PID_Z,  // 104
  PID_Z1  // 105
};
```

It is not hard to see that such modification implies shifting of original OCAF data with inline associating of the shifted PIDs. Furthermore, all existing references must be adjusted correspondingly.

7.4.3.3 Modification

Modification request allows changing Parameter data in a convenient way. With modification request it is not possible to change Parameter type or any other basic Parameter properties.

7.4.3.4 Removal

Removal modification is based on the same approach as insertion and updating ones. I.e. removal is applied on the Sampler Model rather that on OCAF structures directly. Internally it means that the corresponding Conversion Parameter gets specific evolution flag and NULL associated data. This information will be taken into account at applying stage of conversion, where such nullified Conversion Parameters will not be expanded on the resulting OCAF tree.

It is easy to see that removal operation is reference-unsafe. Once any Parameter is deleted, the following normalization actions should be carried out:

- Check whether any back references exist for this Parameter. This can happen if the target Parameter is one of the following types:
  - Plain Reference;
  - Reference List;
  - Tree Function.

  All entries of the target Parameter have to be cleaned up from the detected back reference holders.
- Checks whether the target Parameter is pointed to by any references. The possible reference types are the same as listed above. If any references found, the following rules have to be applied:
  - Plain reference must be nullified;
  - Reference list must be adjusted to throw out target entries;
  - Tree Function must be disconnected;

7.4.3.5 Applying

Once modification requests are gathered, application developer can apply these requests on his data model being converted. The applying stage consists of the following data morphing types:

1. Insertion is applied. Newly added Parameters get temporary dummy PIDs equal to -1;
2. Removal is applied;
3. Modification is applied. Modification is nothing but a simple application of new values for the given Parameter. It is not possible, for instance, to change Parameter type with modification request (you should remove and re-create Parameter in the latter case);
4. Normalization process is performed. Normalization will carry out consistent re-tagging of the final Parameter set, starting from the minimal tag preserved at the very beginning. Newly inserted Parameters will get consistent PIDs instead of their dummy -1 tags. The most important operation here is, however, adjusting of external references.

All these modifications are applied on a copy of the initial data model. In order to produce a copy of OCAF document, we save the passed model to a temporary file and load this file into another model. Using this approach we benefit from OCCT automatic removal mechanism for "ghost" labels. Moreover, the initial data model instance remains unchanged.

7.5 CAF Loader

CAF Loader is a tool dedicated to data model loading operation. Unlike simple opening mechanism provided by standard OCAF services in OCCT, CAF Loader takes care of compatibility conversion. We use a separate class for such loading as compatibility conversion normally requires creation of several data model instances and copying of data between them.

7.6 CAF Dumper

Generally speaking, it is not a simple process to track all changes introduced in the data model in the current development version comparing to its previous version. Even if the Active Data framework remains the same and the only affected thing is the design of a particular data model, the process of domain-specific conversion can be quite sophisticated. In the general case, such conversion affects not only the plain data structures, but also the explicit (Tree Nodes) and implicit (Tree Function Graph Nodes) connectivity of Nodes and Parameters.

In each specific case the developer should possess exhaustive information of the contents of the modification delta to make the conversion routine complete. In order to facilitate the process of gathering such modification deltas, Active Data framework comes with a specific CAF Dumper tool represented by `ActData_CAFDumper` class from `Tools` sub-package. This class allows making textual human-readable dumps of the data model contents. The process of construction of modification delta by dint of CAF Dumper looks as follow:

1. Using the previous version of the application (based on AD framework), developer prepares a non-trivial test project. It is recommended to create such a project in a way that the resulting connectivity between its Data Nodes will be as complex as possible. The latter fact reduces the hypothetical possibility to miss changes in some non-obvious associations. E.g. if your application allows creation of Variable Nodes, it is a good idea to design the test project so that the Variables mechanism is actually used;
2. Developer makes a dump of the prepared test project using CAF Dumper tool. The actual dump will be available as an ASCII human-readable file;
3. Using the recent version of the application, developer prepares just the same test project as in the step 1;
4. Developer makes a dump on the new test project using CAF Dumper tool;
5. Developer compares the contents of these two dump files using a preferred diff utility. Thus one is able to see what has been actually changed in the data model structure. Such diff will guide one through the development of a particular conversion routine which will programmatically apply the manually detected modification delta.

The described process is a formalization of a posteriori approach to elicitation of modification deltas between different versions of data models. Anyway, the ultimate goal here is to collect information about the introduced changes in order to build the conversion routine. Therefore, the simpler (but less secure) way of just keeping the changes in mind can be also reasonable in primitive cases.

Note

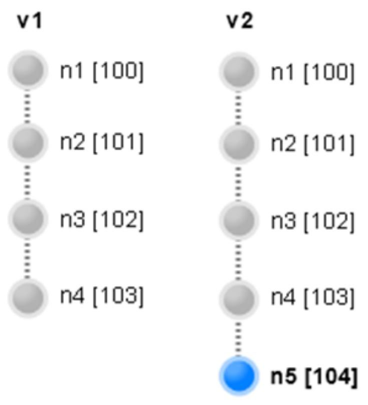
> Please, do not forget also about a-priori way of tracking modification deltas. Such direct tracking can be conveniently implemented via unit tests. Such tests can attempt to load the former versions of your specific projects in the actual environment. Obviously, such tests will fail once the version of the framework or/and application is incremented. Therefore, such scheme keeps you informed that any changes in the data model (even in case of simple version incrementing) must be accompanied with implementing of the correspondent conversion routine. Moreover, as version grows, the test database must be extended with the project files of the older versions. Maintaining such an approach carefully will save your time significantly and can save you from using CAF Dumper challenge in many cases. Just update the tests each time you affect the data model. However, if technically the data model is converted successfully, it does not mean that the conversion is really correct and accurate (some problems can appear in run-time due to insignificant difference in data).

7.7 Unit tests

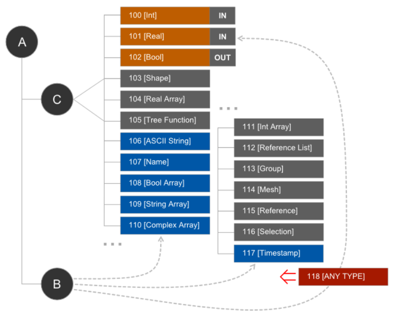
7.7.1 Insertion and appending

7.7.1.1 Case INS-001: simple appending

In this scenario new Parameter n5 is appended to a particular Node class. Other Parameters are not affected. No references are checked in this case.

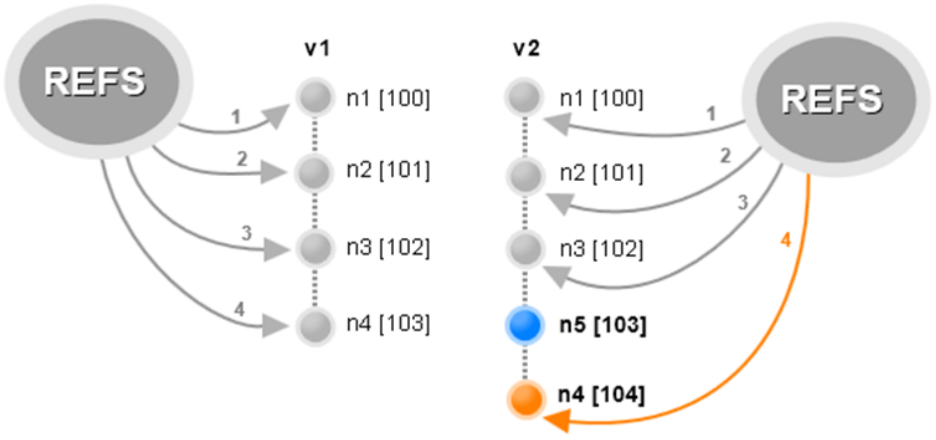


This test is performed on ABC-01 model. New Parameter with PID 118 (first non-occupied one) will be appended to the working Node of type C. Test checks all reasonable Parameter types attempting to append each one for a separate instance of data model.



7.7.1.2 Case INS-002: ordinary insertion

In this scenario new Parameter is inserted before n4.

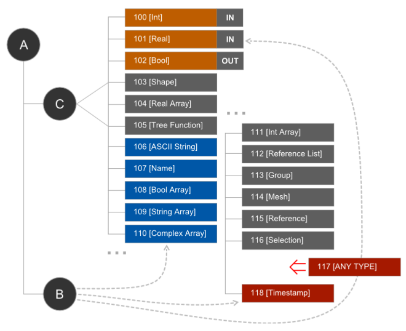


The following peculiarities are checked:

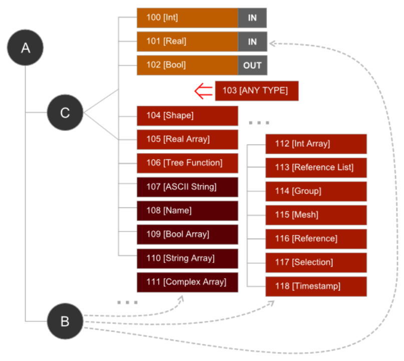
- Parameter n5 gets PID of Parameter n4 (103).
- Parameter n4 is shifted with a new PID (104).
- All references and back-references to Parameter n4 are adjusted so that to refer to its new PID.

This test is performed on ABC-01 Model and divided on several sub-tests:

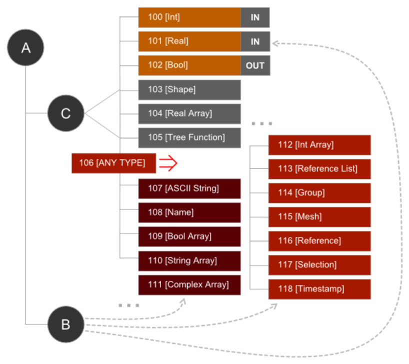
- (a) New Parameter is inserted before existing Timestamp Parameter. The latter one is expected to get PID equal to 118 instead. The only reference established by means of Reference Parameter should be adjusted in that case;



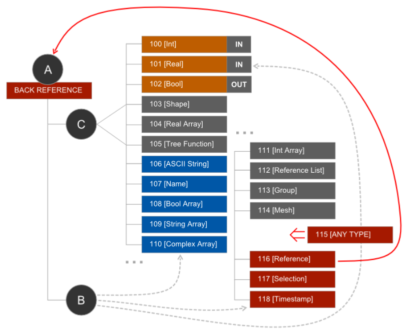
- (b) New Parameter is inserted before existing Shape Parameter. The latter Parameter with all the subsequent ones is expected to be shifted. References established by means of Reference List Parameter and Reference Parameter should be adjusted in that case;



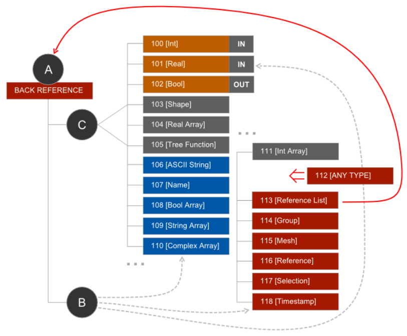
- (c) New Parameter is inserted before existing ASCII String Parameter. The latter Parameter with all the subsequent ones is expected to be shifted. References established by means of Reference Parameter and Reference List Parameter should be adjusted in that case;



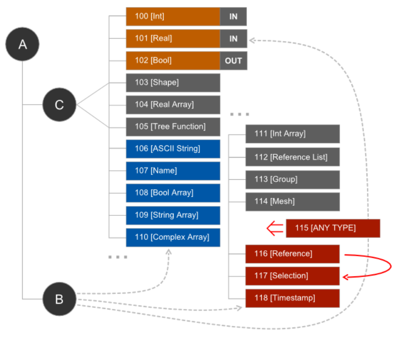
- (d) New Parameter is inserted before initialized Reference Parameter pointing out from Node C. As the latter Reference Parameter will be moved, the corresponding external back-references to it should be adjusted. The Reference Parameter itself should keep its target value unchanged;



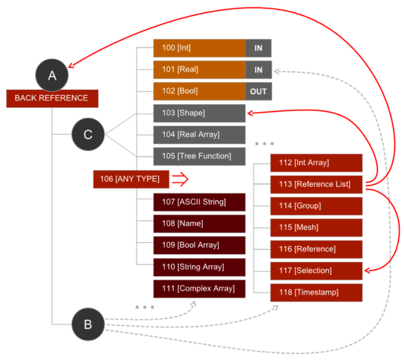
- (e) New Parameter is inserted before initialized Reference List Parameter pointing out from Node C. As the latter Reference List Parameter will be moved, the corresponding back-references to it should be adjusted. The Reference List Parameter itself should keep its target values unchanged;



- (f) New Parameter is inserted before initialized Tree Function Parameter pointing out from Node C. As the latter Tree Function Parameter will be moved, the corresponding back-references to it should be adjusted. The Tree Function Parameter itself should keep its argument and result Parameters unchanged. Moreover, its associated GUID should be preserved;
- (g) New Parameter is inserted before initialized Reference Parameter pointing to the moved Parameter of the same Node. As target of the Reference Parameter will be moved, its value should be adjusted correspondingly. No back references exist in such case;



- (h) New Parameter is inserted before initialized Reference List Parameter pointing to the moved and not moved Parameters. As some targets of the Reference List Parameter will be moved, these targets should be adjusted correspondingly in the converted Model. For out-scoped targets, the corresponding back references should be normalized as well;

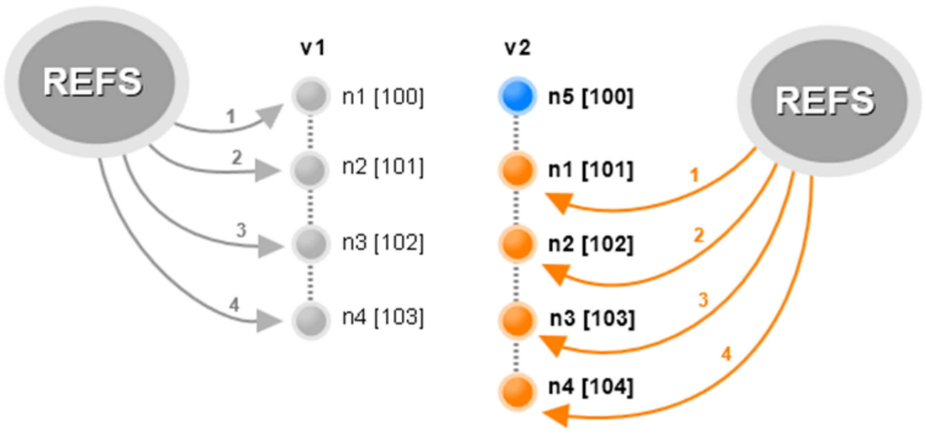


- (i) New Parameter is inserted before initialized Tree Function Parameter with references to moved and not moved Parameters. As some results and arguments of Tree Function Parameter are moved, these results and arguments should be adjusted correspondingly. For out-scoped targets, the corresponding back references should be normalized as well;
- (j) Several append operations. Check that if several insertion records are passed with PIDs equal to -1, Conversion Context will not filter them out according to its "request uniqueness" criterion;
- (k) Check if Conversion Context prohibits twice removal, removal & modification on the same PID, twice modification.

Most of the described tests are performed with single Integer Parameter being inserted. This allows us to focus the unit tests on normalization use cases, rather than on insertion operation itself. The latter operation is exhaustively checked by previous test function.

7.7.1.3 Case INS-003: prepending

This scenario illustrates the "worst" case of insertion as all references have to be adjusted here.

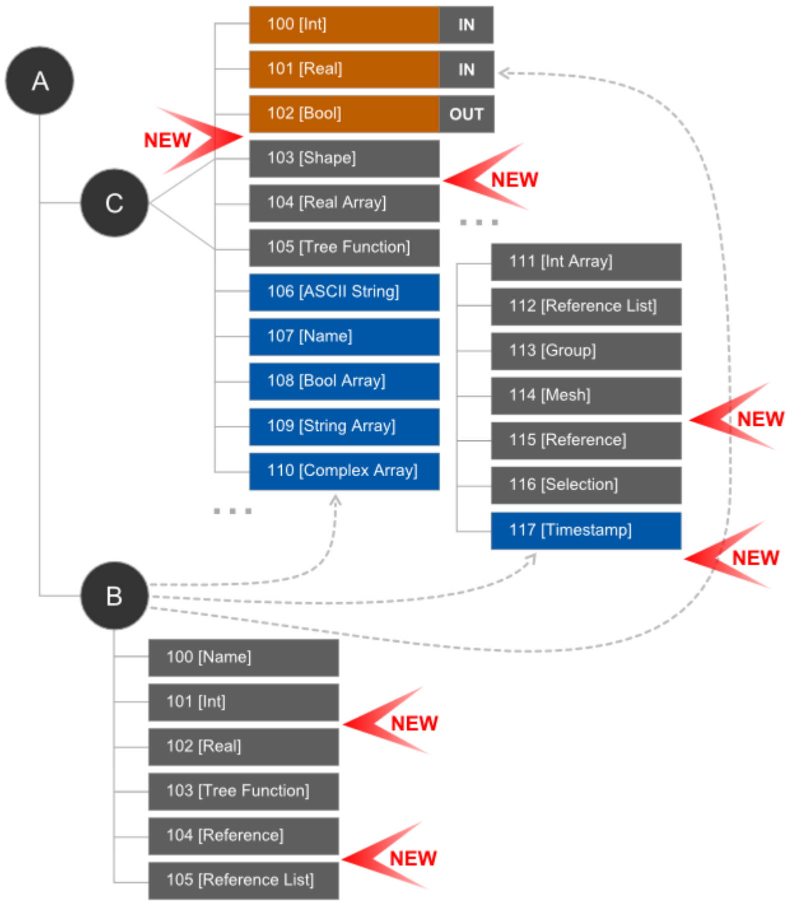


In this case Parameter n5 is inserted before the first Parameter n1. All existing sibling Parameters are shifted in the underlying OCAF tree getting the corresponding PIDs. Notice that Parameter n1 will reuse former PID of Parameter n2 and so on. The corresponding references will be adjusted as well.

This test is performed on ABC-01 model. New Parameter with undefined PID (-1) is inserted before existing Integer Parameter, which is the first one for the Node C. The latter Parameter with all the subsequent ones is expected to be shifted. References established by means of Reference Parameter, Reference List Parameter and Tree Function Parameter should be automatically adjusted in that case.

7.7.1.4 Case INS-004: cumulative insertion test

This test applies several insertion requests on both Nodes C and B for test model ABC-01. The initial state with desired insertion positions is illustrated on the scheme below:



All references should be automatically normalized in both B and C Nodes.

7.7.2 Modification

All unit tests are performed on test model ABC-01.

7.7.2.1 Case MODIF-001: simple test on modification

This test checks all reasonable Parameter types changing their DTOs.

7.7.2.2 Case MODIF-002: type safety

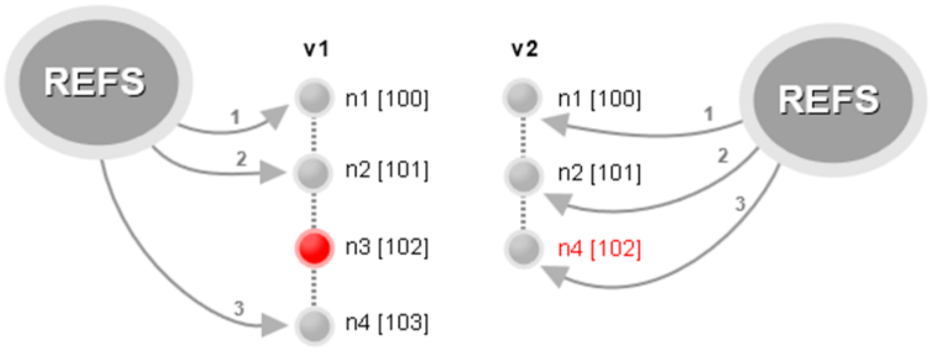
Attempts to change Parameter with DTO of different type. Conversion routine will fail.

7.7.3 Removal

Unit tests are performed on test Model ABC-01. In all cases the resulting Parameters tagging should be contiguous. No "ghost" references should remain.

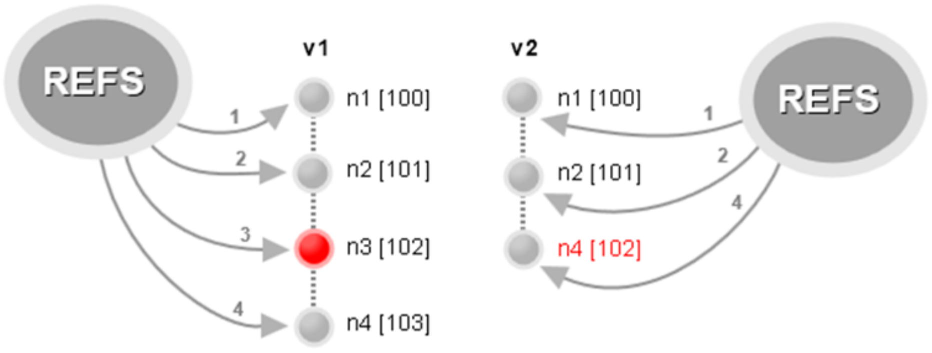
7.7.3.1 Case REMOVE-001

Some non-reference Parameter without any references is removed. Re-tagging and normalization should perform correctly.



7.7.3.2 Case REMOVE-002

Plain reference exists for the target non-reference Parameter. Target Parameter has to be dropped with re-tagging of the remaining ones. Existing reference should be nullified. Existing back-reference corresponding to the nullified reference should be normalized.



7.7.3.3 Case REMOVE-003

Reference list pointing to the target non-reference Parameter exists. Target Parameter has to be dropped with re-tagging of the remaining ones. Existing reference list should be normalized, so that all target entries have to be cleaned up. Existing back-reference should be normalized.

7.7.3.4 Case REMOVE-004

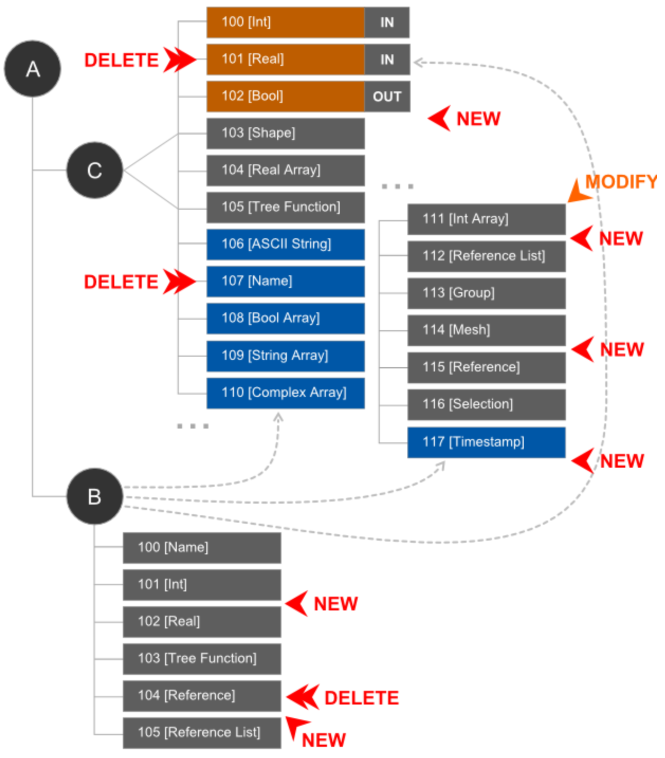
Tree Function pointing to the target non-reference Parameter exists. Target Parameter has to be dropped with re-tagging of the remaining ones. Existing Tree Function should be disconnected.

7.7.3.5 Case REMOVE-005

Plain reference exists for non-empty Reference List Parameter pointing to inside and outside of the target Node.

7.7.4 Complex modification

The cumulative test works on the following initial data model:



To make this validation case complete, test function uses specific Data Node Cursor representing the converted data object. This Data Node is settled down onto the produced OCAF tree in order to check its WELL-FORMED state.

Chapter 8

Room for Improvement

8.1 Common

- Sample application for Active Data;
- Update to standard OCCT mesh attribute once implemented;
- Review possibility to merge ActiveDataAPI and ActiveDataAux with the main ActiveData library;
- Get rid of matrix parameters as they are not efficient. Or consider introducing 2-dimensional arrays as OCAF attributes;
- Format of AD is very verbose. Can we reduce the amount of meta-information around Parameters?
- Estimate memory footprint and performance.

8.2 Copy/Paste

- Implement Cut operation as an alternative to Paste. This might be especially useful when large data sets are copied. In the latter case it could be reasonable not to keep the buffer at all;
- Multi-buffering;
- Copy/Paste buffer is saved to a file along with other data. This is, however, redundant but we cannot just remove the buffering section before saving as it will drop the buffer. The best option would be ignore the buffering section (along with LogBook section too) when saving, however, it is not clear not how to do it.

8.3 Undo/Redo

- Elaborate adaptive Undo limitation mechanism. E.g. it is a good idea to establish limitation rules such as number of steps and maximum occupied memory;

8.4 Compatibility Conversion

- Improve CAF Dumper, so that to allow dumping not only the structure of data model, but also Dependency Graphs;
- Improve Conversion Context with possibility to replicate any given atomic operation for the given type of the Node. Without this possibility application developer is forced to iterate over the entire model in order to apply the same kind of data morphing to each Node instance;

- Gather conversion statistics;

Chapter 9

APPENDIX: Comparison with TObj

TObj is another framework shipped with Open CASCADE Technology and providing means for designing your data models. Here we discuss the differences between TObj and Active Data. Such discussion is necessary in order to have full overview on the available OCAF-based tools.

1. Separation between user tree and storage tree. In TObj the storage hierarchy is exactly the same as user hierarchy (`TDataStd_TreeNode` attribute is not used). In Active Data the storage hierarchy has nothing to do with the user hierarchy. TObj approach is more straightforward and does not require any sophisticated tools for copy/paste;
2. In Active Data there is no need to work with entities like `TDF_Label` at all. All OCAF internals are completely hidden from programmatical point of view. In TObj you need to have a `TDF_Label` to create a new object. Moreover, you have to descend to bare OCAF in order to set non-standard attributes which are forbidden in Active Data;
3. Sub-models are not supported as a concept in Active Data. TObj, however, comes with such a principle;
4. TObj intensively uses custom OCAF attributes, while Active Data provides a basis set of property types to express any kind of type. Therefore TObj requires its own storage/retrieval drivers while Active Data can work with the standard ones;
5. Active Data allows to bind application-specific flags without disclosing their actual nature. TObj comes with standard "Object State" flags containing "hidden", "saved" etc;
6. TObj is a passive data framework, while Active Data is designed to support `TFunction` mechanism and enables parametric dependencies so;
7. TObj is easier to study;
8. Active Data comes with compatibility conversion tools which allow adding/removal of object properties without a need to reserve space for "future extensions";
9. Active Data comes with `TFunction`-based mechanism of global variables.

To summarize, Active Data framework has the following main differences comparing to TObj:

1. Activeness (`TFunction` mechanism enabled + variables mechanism inside);
2. Standard OCAF under the cover;
3. Compatibility conversion utilities;
4. More flexible in terms of relations between objects;
5. Active Data is rather a *practice* of using OCAF, while TObj is an abstracted OCAF-based product.

Chapter 10

References

1. http://dev.opencascade.org, official development portal : OPENCASCADE SAS
2. http://dev.opencascade.org/doc/overview/html/index.html : official documentation, OPENCASCADE SAS
3. http://isicad.net/articles.php?article_num=17368 : "Open CASCADE Technology Overview", OPENCASCADE SAS

Chapter 11

Todo List

Page Mesh

This section will describe custom Mesh Attribute available in Active Data framework. Currently this description is not available.

Page Tree Functions

SpyLog functionality still needs to be described.