EXPRESS Data Manager™

C++ EXPRESS API

User's Guide

Document id.	C++ EXPRESS API Users Guide
File name	C++ EXPRESS API Users Guide.doc
Doc version	V 4.1
Status	Final
Date	2020-04-01

TABLE OF CONTENT

TABLE OF FIGURES

TABLE OF TABLES

1. Introduction

The EDM C++ EXPRESS API is a C++ library for handling EXPRESS modeled data stored in an EDM database. The main characteristics of the API are:

- Early binding data definitions - *EDMsupervisor*™ has a generator that generates C++ classes that represent the entity and defined types in the EXPRESS schema in question.
- Designed to handle simple as well as the most comprehensive EXPRESS schemas defined so far.
- Two different memory models:
  - C++ Objects in memory. The C++ class representations of the EDM database instances must be read into memory before they can be used by the program. New objects are created in memory and written to the database later by the method *writeAllObjectsToDatabase*. Programs must declare
    ```
     using namespace OIM;
    ```
  - C++ Objects in the database. The C++ Objects are handles to objects in the database. The get and put methods for the attributes of the objects are using *EDMinterface*™ to read and write attribute values. To use this memory model one must compile the project with the preprocessor directive *USE_EDMI*. Programs must declare
    ```
    using namespace OID;
    ```
- The use of a memory allocator (*CMemoryAllocator)* that provides efficient memory management is mandatory.
- The generated C++ classes do not have destructors. This implies that it is not possible to delete (release memory) of single objects. Objects are released collectively by deleting the *CMemoryAllocator* object or by executing the *Reset* method of the *CMemoryAllocator* class. The benefit of this way of deleting objects is better performance and almost no probability for memory fragmentation.

A project using the C++ EXPRESS API is started with generating a C++ header file (.hpp) and a corresponding implementation (.cpp) file from the EXPRESS schema that shall be used in the project.
The *EDMsupervisor*™ has the command Schemata->Generate Interface->Cpp 2010 for this purpose. The header file will contain one C++ class for each entity and one type definition for each type in the EXPRESS schema.

2. Change log

Changes from version 2.0 to 3.0:

1. Added chapter 4. EDM Server Side C++ Plug-in
2. Corrected the code example in 3.12
3. Corrected text before using namespace ccpm in chapter 3.4.1
4. Explained how to declare the EXPRESS schema object in the example in chapter 3.4.1
5. Modified a comment in the 3.6.2 example
6. Added missing methods for Methods for Bag, Set and List in chapter 3.7
7. Modified chapter 3.11
8. Rephrased chapter 3,12
9. Chapter 3.13: The SubModel concept is abandoned, since the “thin” memory model should be used in case of memory shortage.
10. Remade chapter 3.14
11. Improved the description of *edmiRemoteExecuteCppMethod* in chapter 4.
12. Added more about complex entities in chapter 5.
13. Added content to chapter 5.2 (Use of *EDMinterface*™)
14. Reworked 3.6.1
15. Substituted example in chapter 3.7, since the old one was erroneous.
16. Remove chapter 3.7.1
17. Substituted example in chapter 3.8
18. Extended the example in chapter 3.9
19. Substituted the examples in chapter 3.11
20. Substituted the clone example in chapter 3.8
21. Chapter 6.1 : *EXPRESS Data Manager*™ installation no longer contains cpp10 examples
22. Substituted the first example in chapter 3.10
23. Chapter 6 substituted with Appendix A and B.

Changes from version 3.0 to 3.1:

1. Added information about how to download the software.

Changes from version 3.1 to 3.2:

1. Changed namespace names from cppm/cppd to OIM/OID.

3. C++ EXPRESS API - basics

3.1 Mapping between EXPRESS and C++

The different types in an EXPRESS schema are mapped to C++ in the following way:

Table 1: Mapping of EXPRESS constructs to C++

EXPRESS	C++
Entity	Class The attributes are accessed via put and get methods.
Complex entity	Class with name equal to the complex entity name except that the '+' delimiter is substituted with '_'.
Entity with multiple inheritance	Class with multiple inheritance
Select	If all the alternatives of the select are entities, it is mapped to dbInstance which is the common superclass for all C++ classes representing entities. Otherwise the select is mapped to cppSelect which is identical to the select struct used in SDAI.
Enumeration	typedef enum {.., .., } <Enumeration name>;
Integer	EDMLONG
Real	Double
Logical	`typedef enum {logicalFalse = 0, logicalUnknown, logicalTrue} logical;`
Boolean	Bool
String	char *
Array	`template <typename ArrayElement>` `class Array : public dbArray`
List, Set and Bag	Subtypes of class dbLinkedAggregate. For List the declaration is as follows: `template <class ListMember>` `class List : public dbLinkedAggregate` `Example:` `EXPRESS:` `ENTITY applied_action_assignment` `SUBTYPE OF (action_assignment);` `items : SET [1:?] OF action_items;` `END_ENTITY;` `C++:` `class applied_action_assignment : public action_assignment` `{` `public:` `Set<action_items*>*   get_items();` `void                   put_items(Set<action_items*>* v);` `void                   put_items_element(action_items*);` `};`

The C++ classes have, in addition to constructors, put and get methods for each explicit attribute. Derived and inverse attributes have only get methods as their values are computed and set automatically.
Every C++ class that represents an EXPRESS entity inherits the class dbInstance. The dbInstance class refer to the attribute buffer where attribute values are stored, and contains the unique object identifier in the EDM database. When a C++ object is created in the client memory, the unique database object identifier is set to NULL. When the C++ EXPRESS API has created a corresponding object in the EDM database, the unique database object identifier gets the correct value from the database.
Using the *OIM* (objects in memory) pattern of the C++ EXPRESS library, it is possible to link objects together forming deep structures in the client memory before storing the objects in the database. At a certain point in time one can store all objects in the database with one single function call. The inverse attributes are not assigned when using *OIM* pattern. That will happen when the objects are written to database, and when the objects are read back to memory at a later stage, the inverse attributes are correctly assigned. The same pattern applies for the derived attributes.

3.2 Memory management for the OIM pattern

The C++ EXPRESS API is designed to handle large amounts of data very efficiently. To meet this requirement, memory for the C++ objects is managed by the *CMemoryAllocator* class. The *CMemoryAllocator* class uses the malloc() function to allocate chunks of memory from the C++ runtime system whenever needed. The parameter in the constructor for *CMemoryAllocator* specifies the chunk size. In the following example a *CMemoryAllocator* object is created with default chunk size of 1000.000 bytes:

```
    CMemoryAllocator* ma = new CMemoryAllocator(1000000);
```

The generated C++ classes do not have destructors. This implies that it is not possible to delete (release memory of) single objects. Objects are released simultaneously by deleting the *CMemoryAllocator* object or by executing the *Reset* method. The benefit of this way of deleting objects is better performance and almost no probability for memory fragmentation.

3.3 Memory management for the OID pattern

In addition to the memory management scheme described in the section above, the C++ EXPRESS API has a version that use *EDMinterface*™ to operate on the objects in the database directly. The applications memory consumption is considerably reduced in this case (objects are stored in the database cache and swapped on demand). Nevertheless the *CMemoryAllocator* class is applied as well when using the *OID* pattern. To use this option C++ programs must be compiled with the Pre-processor Directive *USE_EDMI* defined and linked with cpp10_edmi.lib instead of cpp10.lib.

3.4 Database, Repository, Model, Schema and EDMI

A set of populated EXPRESS objects in an EDM database is called a model. EDM models are grouped together in repositories and a database consists of several repositories. To operate on an EDM database the *EXPRESS Data Manager*™ offers a rich library of functions called EDMI, that is described in *EDMassist.*
http://edmserver.epmtech.jotne.com/EDMAssist/WebHelp/EDMAssist.htm.
The C++ EXPRESS API is designed to store objects in and retrieve objects from an *EXPRESS Data Manager*™ (EDM) database. In order to ease the storage and retrieval of C++

objects to/from the EDM database, the C++ EXPRESS API have C++ objects representing Database, Repository and Model. See the following example:

```
CMemoryAllocator ma(1000000);
// Database object is declared
Database db("database\directory", "database_name", "password");
// Database is opened by implicite call to EDMI
db.open();
// Repositor is declared
Repository cRepository(&db, "DataRepository");

// myModel is declared with reference to the underlying schema.
// The memory allocator assignes memory to the objects in the model.
// my_Schema is a reference to the generated C++ representation
// of the underlying EXPRESS schema. There are naming conventions for the //
generated EXPRESS schema object, and the declaration shall be like:
// <name space>_Schema my_Schema = <name_space><schema name>_SchemaObject,
// where <name_space> is given when you generate C++ classes by means of
// the EDMSupervisor™  .
E.g.
ap209_Schema my_Schema =
ap209_multidisciplinary_analysis_and_design_mim_lf_SchemaObject;
Model myModel(&cRepository, &ma, &my_Schema);
// nyModel is opened for read and write
myModel.open("myModel", sdaiRW);
```

3.4.1 Create objects in memory

In the C++ EXPRESS API objects are created with overloading of the *new* operator. Each new object must belong to a Model, for this reason the new operator takes a reference to a Model object as a parameter. The attributes of new objects are stored in memory by the generated *put* methods without copying the values to the database. All created objects with their attributes can be written to the database model by means of the method

```
    Model::writeAllObjectsToDatabase();
```

To create objects in memory the application source must contain the following declaration:

```
        using namespace OIM;
```

Figure 1 shows a sketch where three objects of the AP209 entities product, product_definition_formation and product_definition are linked together to form the data structure for an AP209 product. The following C++ code sequence exemplifies how the objects are created, how the attributes of the objects are set and how the objects are linked together.

```
#define newObject(className) new(m)className(m)

Model *m = &myModel;

// Create product object and set its attributes
product* p1 = newObject(product);
p1->put_id("Aircraft 1");
p1->put_name("Aircraft propeller map");
p1->put_description("Aircraft 1");

// Create product_definition_formation object
product_definition_formation* pdf = newObject(product_definition_formation);
pdf->put_description("Aircraft with two front doors and wheel landing gear");
pdf->put_id("1");
// Link product_definition_formation object to product object
pdf->put_of_product(p1);
```

```
// Create product_definition object
product_definition* pd = newObject(product_definition);
pd->put_id("pd-1");
pd->put_description("description");
// Link product_definition object to
// product_definition_formation object
pd->put_formation(pdf);
// and update the database
m->writeAllObjectsToDatabase();
```

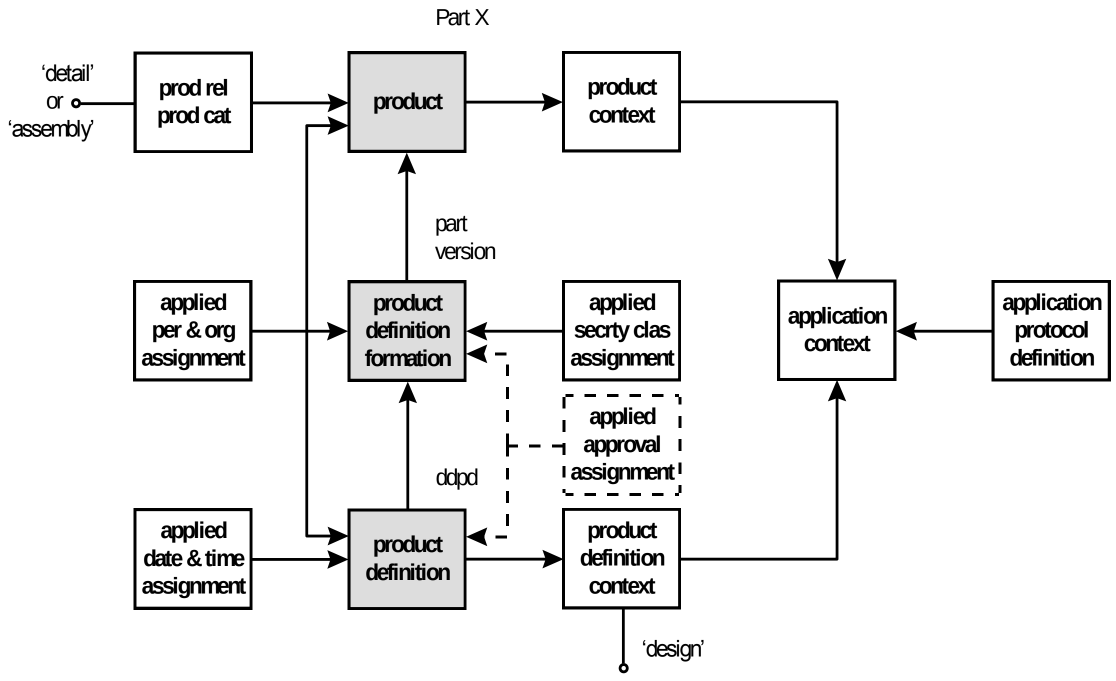

Figure 1 – Diagram of a small AP209 instantiation (source: AP209ed2 recommended practices)

3.4.2 Create objects in database directly

With the Pre-processor Directive *USE_EDMI* defined, C++ objects are created directly in the database. If you use the constructor without database object id (InstanceId), an object will be created in the database. If you use the other constructor with database object id, the C++ object will be a reference to an already existing object in the database. To create objects in the database directly the application source must contain the following declaration:

```
      using namespace cppd;
```

The example in the previous section will then look like:

```
Model *m;
// Create product object and set its attributes
product p(m);
p.put_id("Aircraft 1");
p.put_name("Aircraft propeller map");
p.put_description("Aircraft 1");

// Create product_definition_formation object
product_definition_formation pdf(m);
pdf.put_description("Aircraft with two front doors and wheel landing gear");
pdf.put_id("1");
// Link product_definition_formation object to product object
```

```
Pdf.put_of_product(&p);

// Create product_definition object
product_definition pd(m);
pd.put_id("pd-1");
pd.put_description("description");
```

3.5 Enumeration

EXPRESS enumerations are mapped to C++ enum types where the enumeration values are prefixed with the name of the enumeration:

<enumeration name>_<enumeration value (in uppercase)>

Example:

EXPRESS:

```
TYPE enumerated_curve_element_freedom = ENUMERATION OF (
    x_translation, y_translation, _translation,
    x_rotation, _rotation, _rotation,
    warp,
    none );
END_TYPE;

C++:
typedef enum {enumerated_curve_element_freedom_X_TRANSLATION,
       enumerated_curve_element_freedom_Y_TRANSLATION,
       enumerated_curve_element_freedom_Z_TRANSLATION,
       enumerated_curve_element_freedom_X_ROTATION,
       enumerated_curve_element_freedom_Y_ROTATION,
       enumerated_curve_element_freedom_Z_ROTATION,
       enumerated_curve_element_freedom_WARP,
       enumerated_curve_element_freedom_NONE}
enumerated_curve_element_freedom;
```

3.6 Select

It is not possible to map select types directly to C++. In the C++ EXPRESS API selects are treated in two different ways:

- If all the alternatives in the select type are entities (object classes), the select is treated as an entity.
- If the select have at least one alternative different from entity, the select will be declared as a class cppSelect, which is the same way as selects are treated in SDAI.

3.6.1 All select alternatives are entities

The select is mapped to the supertype of all entities namely dbInstance. This will create efficient code, but has the drawback that every object is applicable where such selects are specified. When such selects return values in, for example, get-functions, the generated C++ type of the return value is void *. The void * must then be casted to the correct C++ object pointer. Example:

```
TYPE characterized_definition = SELECT (
    characterized_object,
    characterized_product_definition,
    shape_definition);
END_TYPE;
```

The generator will generate the following for characterized_definition:

```
typedef dbInstance characterized_definition;
```

The entity property_definition has an attribute of type characterized_definition with name definition. All the alternatives of characterized_definition are entities.

As an example, we create a property and connect it to the product_definition defined above. In this case we know that a product_definition is attached to the property, but in the general case we do not know this, and, therefore, we exemplify how all possible entity types in the graph of the select type characterized_definition are handled.

```
// Create a property
   property_definition *prop_def_1= newObject(property_definition);
   prop_def_1->put_name("Prop1");
// set the definition attribute beeing the product definition from above
   prop_def_1->put_definition(pd);
// Retrieve the proprty definition and exemplify how is processed
// based on the retrieved type.
   entityType objectType;
   void* obj = prop_def_1->get_definition(&objectType);
   if(objectType == et_product_definition){
     product_definition *mypd = (product_definition*)obj;
     product *myp = mypd->get_formation()->get_of_product();
     printf("\nProperty 1 is connected to product: %s",myp->get_id());
   } else if(objectType == et_characterized_object){
   // handle characterized_object
   } else if(objectType == et_product_definition_shape){
   // handle product_definition_shape
   } else if(objectType == et_shape_aspect){
   // handle shape_aspect
   } else if(objectType == et_shape_aspect_relationship){
   // handle shape_aspect_relationship
  } else if(objectType == et_product_definition_relationship){
   // handle product_definition_relationship
  } else {
    throw new CedmError(sdaiETYPEUNDEF);
  }

   // Create another property and attach it to a characterized_object
   property_definition *prop_def_2= newObject(property_definition);
   prop_def_2->put_name("Prop2");
   characterized_object *co = newObject(characterized_object);
   co->put_name("co1");
   prop_def_2->put_definition(co);

// Retrieve the proprty definition and exemplify how is processed
// based on the retrieved type.
   obj = prop_def_2->get_definition(&objectType);
   if(objectType == et_product_definition){
   // handle product_definition
   } else if(objectType == et_characterized_object){
     characterized_object *myco = (characterized_object*)obj;
     printf("\nProperty 2 is connected to characterized_object: %s",myco-
>get_name());
   } else if(objectType == et_product_definition_shape){
   // handle product_definition_shape
   } else if(objectType == et_shape_aspect){
   // handle shape_aspect
   } else if(objectType == et_shape_aspect_relationship){
   // handle shape_aspect_relationship
   } else if(objectType == et_product_definition_relationship){
   // handle product_definition_relationship
   }else {
    throw new CedmError(sdaiETYPEUNDEF);
   }
```

3.6.2 Select where at least one of the alternatives is not an entity

See the EXPRESS example below. That is a select where the alternatives are either a string or an enumeration.

```
TYPE application_defined_degree_of_freedom = STRING;
END_TYPE;

TYPE enumerated_curve_element_freedom = ENUMERATION OF (
    x_translation, y_translation, _translation,
    x_rotation, y_rotation, z_rotation,
    warp,
    none );
END_TYPE;

TYPE curve_element_freedom = SELECT (
    enumerated_curve_element_freedom,
    application_defined_degree_of_freedom);
END_TYPE;
```

The C++ generator generates the following class declaration for the select type curve_element_freedom:

```
class curve_element_freedom : public cppSelect {
public:
   curve_element_freedom() { }
   curve_element_freedom(enumerated_curve_element_freedom v, Model *m) ;
   curve_element_freedom(application_defined_degree_of_freedom v) ;
};
```

```
// The code below shows how to create curve_element_freedom selects
// and put it into the attribute release_freedom of
//_curve_element_end_release_packet objects
// By new(m), the selects are created in memory controlled Models
// The m after enumerated_curve_element_freedom_X_ROTATION is necessary
// because enumerated_curve_element_freedom_X_ROTATION must be translated
// to the corresponding instance identifier in the underlying database.
curve_element_freedom  *cef1;
cef1 = new(m)curve_element_freedom(enumerated_curve_element_freedom_X_ROTATION, m);
curve_element_freedom  *cef2;
cef2 = new(m)curve_element_freedom("application defined degree of fredom",m);

curve_element_end_release_packet * ceerp1;
ceerp1 = newObject(curve_element_end_release_packet);
ceerp1->put_release_freedom(cef1);
curve_element_end_release_packet * ceerp2;
ceerp2 = newObject(curve_element_end_release_packet);
ceerp2->put_release_freedom(cef2);
```

3.7 The aggregates Bag, Set and List

The aggregate types bag, set and list are implemented as subtypes of dbLinkedAggregate. Since these aggregates are expandable, they have a common implementation. Arrays, that have fixed sized, have another implementation.
The type of the aggregate element is added by the C++ template mechanism. List is declared as follows:

```
template <class ListMember>
class List : public dbLinkedAggregate
```

For example the EXPRESS type LIST OF INTEGER will then be declared as

```
    List<int>
```

Aggregates are allocated in memory by a new operator that requires reference to a memory allocator object. Bag, set and list aggregates have constructors with 3 parameters:

1. Reference to memory allocator object.
2. Constant specifying the primitive type of the aggregate element. The primitive type must be specified by one of the following values:
    a. sdaiINTEGER
    b. sdaiREAL
    c. sdaiBOOLEAN
    d. sdaiLOGICAL
    e. sdaiSTRING
    f. sdaiENUMERATION
    g. sdaiINSTANCE
    h. sdaiAGGR
3. An integer specifying the size, in number of aggregate elements, of each of the buffers linked together to store the aggregate in memory. **Note** that this does not limit the total size of the aggregate.

A List<int> is then allocated in memory as shown below

```
// The first 3 ints are stored in the first buffer and
//the next 3 ints in the next buffer etc.
    List<int> intList = new(ma)List<int>(ma, sdaiINTEGER, 3);
```

Methods for Bag, Set and List:
The current version of the C++ EXPRESS API has the following methods for Bag, Set and List:

- void add(element, CMemoryAllocator) – adds the specified element to the aggregate. Reference to memory allocator object is needed because adding an element may imply allocating a new buffer for the element.
- int size() – return number of elements in the aggregate.
- void reset() – number of elements is set to zero without releasing the buffers
- ListMember first() – return the first element of a list
- ListMember getElement(EDMULONG elementNo) – return list element by index
- double getRealElement(EDMULONG elementNo) – return real value by index

```
SET example:
// Create another product
  product* p2 = newObject(product);
  p2->put_id("Aircraft 2");
  p2->put_name("Aircraft propeller map");
  p2->put_description("Aircraft 2");
  printf("\nProduct Aircraft 2 created.");

// Allocate SET OF products
  Set<action_items*>* prodSet = new(&ma) Set<action_items*>(&ma, sdaiINSTANCE, 2);
// Add the two products already created
   prodSet->add(p1,&ma);
   prodSet->add(p2,&ma);
// Assign set to attribute items of applied_action_assignment
   applied_action_assignment *aaa = newObject(applied_action_assignment);
   aaa->put_items(prodSet);

LIST example :
  List<STRING> theWeekDays(&ma, sdaiSTRING, 7);
  theWeekDays.add("Sunday", &ma);
  theWeekDays.add("Monday", &ma);
  theWeekDays.add("Tuesday", &ma);
  theWeekDays.add("Wednesday", &ma);
```

3.8 *Aggregates of aggregates*

How to handle aggregates of aggregates in the OID object model is demonstrated through the following example:

The example has this Express schema:

```
TYPE Language = LIST[1:2] OF STRING;
END_TYPE;

ENTITY ContentProperty;
  languages : OPTIONAL SET[1:?] OF Language;
END_ENTITY;
```

```
// Initiate a ContentProperty with two languages, each with three words.

Model *m = ...
ContentProperty *cp = new(m)ContentProperty(m);
Set<Language> languages(m);
// An aggregate must be stored as an attribute in the database before
// elements can be added to it.
cp->put_languages(&languages);

Language norwegian(m), english(m);
languages.add(&norwegian);
languages.add(&english);
norwegian.add("ja"); norwegian.add("nei"); norwegian.add("ikke");
english.add("yes"); english.add("no"); english.add("not");
```

```
// Loop through the languages of a ContentProperty and print their words.

ContentProperty *cp = ...

if (cp->exists_languages()) {
   Iterator<Language*, cpp10::entityType> langIter(cp->get_languages());
   for (Language *l = langIter.first(); l; l = langIter.next()) {
      printf(”\n”);
      Iterator<char *, cpp10::entityType> stringIter(l);
      for (STRING s = stringIter.first(); s; s = stringIter.next(), i++) {
         printf(”   %s\n”, s);
      }
   }
}
```

3.9 *Iterators*

To access the individual members of a Bag, Set or List, the C++ EXPRESS API offers the *Iterator* concept. An *Iterator* is a data structure that keeps track of the current position when the program loops trough an aggregate. The aggregate is input parameter to the *Iterator* constructor.
An *Iterator* has a *first* method that returns the first element of an aggregate. After getting the first element of an aggregate, one advances to the next element by means of the *next* method. After getting an aggregate element by either the *first* or the *next* method, one must check if the iterator is exhausted. For pointer elements like object, aggregate or select, one simply checks if the pointer is different from NULL. For elements that not are pointers, like Enumeration, Integer, Real, Logical or Boolean one must use the method *moreElems*. The example below shows the pointer case.

```
Example : print product ids from a set of products.

   Set<product*>* prodSet =....
   // Iterator for the set of products
   printf("\n\nProducts in set:");
   Iterator<product*, entityType> prodIter(prodSet);
   int i =0;
   for(product *prod = prodIter.first(); prod; prod = prodIter.next()){
      ++i;
      printf("\nElement : %d - %s", i, prod->get_id());
   }
```

3.9.1 Execution time improvement.

In the iterator's first() and next() methods, the returned product pointer points to an object created inside the method. If the set of product is big, this can be a time consuming part of the execution that can be reduced. By declaring the C++ product object on the stack before the iterator and retrieving the object id rather than the object itself, the execution time can be reduced. See the example below:

```
   Model *m = ..
   Set<product*>* prodSet = ..
   // Declare a product object on the stack
   product myProd(m, 0);
   printf("\n\nProducts in set:");
```

```
    // Iterator for the set of products
    Iterator<product*, entityType> prodIter(prodSet);
    int i =0;
    entityType prodType; // the actual object type is returned from first/next
                         // (only relevant if product has subtypes).
    for(SdaiInstance prodId = prodIter.firstId(&prodType); prodId;
       prodId = prodIter.nextId(&prodType)) {
       myProd.setObjectId(prodId);
       ++i;
       printf("\nElement : %d - %s", i, myProd.get_id());
    }
```

3.10 Array

Arrays are implemented in a way similar to the other aggregates. The difference is that an array has fixed size while the others are expandable. The basic array class is dbArray, and the array element type is added by the template mechanism.

```
        template <typename ArrayElement>
        class Array : public dbArray
```

An array of object pointers is declared in the following way:

```
   Array<product*> *prodArr = new(ma)Array<product>(ma, sdaiINSTANCE, 3, 8);
```

The parameters are the same as for the other aggregates, except for that the buffer size parameter is substituted with *Min* and *Max* boundaries of the array.

Array methods:

- ArrayElement get(int index) - returns the array elements at the specified index.
- void put(int index, ArrayElement am) - puts an array element at the specified index.
- void unSet(int index) – unset an array element at the specified index.
- bool isSet(int index) – returns true if the element at the specified index is set otherwise false.
- `ArrayElement& operator[] (int index) –` operator that works for the types sdaiINTEGER, sdaiREAL, sdaiBOOLEAN and sdaiLOGICAL. This method is only available for the in memory object model.

Examples:

```
// Declare array of int with min index 5 and max index 15
Array< EDMLONG> arr(&ma, sdaiINTEGER, 5, 15);
// Initialize the array
for (EDMLONG i=5; i <= 15; i++) arr[i] = 100 + i;
// Print the array values
for (EDMLONG i=5; i <= 15; i++) {
   printf("arr[%d] = %d\n", i, arr[i]);
}

// An example using ap209 euler_angles
   parametric_volume_3d_element_coordinate_system *pv3d =
newObject(parametric_volume_3d_element_coordinate_system);
   Array<double>angles (&ma, sdaiREAL, 1, 3);
   angles[1] = 2.7; angles[2] = 3.14; angles[3] = 6.6;
   euler_angles *euang = newObject(euler_angles);
   euang->put_angles(&angles);
   pv3d->put_eu_angles(euang);
```

```
// The entity euler_angles have an array that is retrieved
// by the method ea->get_angles();
   euler_angles *ea = pv3d->get_eu_angles();
   Array<double> *myangles = ea->get_angles();
   double angle1 = myangles->get(1);
   double angle2 = myangles->get(2);
   double angle3 = myangles->get(3);
   printf("\n\nEuler angles : %f,%f,%f",angle1,angle2,angle3);
```

3.11 Multiple inheritance, subtypes and supertypes

Working with objects of classes that have subtypes and supertypes in C++ may be challenging. In the following we will address these challenges.
When one is using iterators to iterate over aggregates, the *first* and *next* methods returns the pointers as they were added to the aggregate. If the object class of the aggregate element has several subtypes, pointers to objects of the specific subtypes are returned. The following example shows such a situation where the program shall handle the different subtypes .

```
The example below is applicable for the in database memory model.
// Create a representation_context and three objects that all are subtypes of
// representation namely a node, fea_model_3d and explicit_element_representation
representation_context rep_ctx(m);
rep_ctx.put_context_identifier("Context1");
node n(m);
n.put_name("Node 1");
n.put_context_of_items(&rep_ctx);

fea_model_3d feam(m);
feam.put_name("Feam 1");
feam.put_creating_software("Lorentz developed the software");
feam.put_context_of_items(&rep_ctx);

explicit_element_matrix elm(m);
matrix_property_type mpt;
mpt.setString("Property type");
elm.put_property_type(&mpt);
explicit_element_representation eep(m);
eep.put_name("explicit_element_representation 1");
eep.put_matrix(&elm);
eep.put_context_of_items(&rep_ctx);

// Then get the representations attached to the context
// and handle them separately
Set<representation*> *reps;
reps = rep_ctx.get_representations_in_context();
entityType repType;
Iterator<representation*, entityType> repIter(reps);
for (representation* rep = repIter.first(&repType); rep; rep =
repIter.next(&repType)) {
  if(repType == et_node) {
    node *mynode = (node*)rep;
    printf("\nNode : %s",rep->get_name());
  }
  if(repType == et_fea_model_3d) {
    fea_model_3d *myfeam = (fea_model_3d*)rep;
    printf("\nfea_model_3d : %s",rep->get_name());
    printf("\nfea_model_3d software: %s",myfeam->get_creating_software());
```

```
   }
   if (repType == et_explicit_element_representation) {
     explicit_element_representation *myeet =
(explicit_element_representation*)rep;
     printf("\nexplicit_element_representation : %s",rep->get_name());
     explicit_element_matrix *myeem = myeet->get_matrix();
     matrix_property_type *mympt = myeem->get_property_type();
     char *mychar = mympt->getString();
     printf("\nMatrix property type : %s",mychar);
   }
}
```

3.12 Finding references to objects

The EXPRESS function USEDIN returns each object that uses the specified object in a specified role, where the role is <schema_name>.<entity_name>.<attribute_name>. The C++ EXPRESS API has a similar construction. When object A is referencing another object (object B) by assigning object B to the appropriate attribute of object A, a pointer from B back to A is set. In this way it is easy to find all objects that are referencing object B. In other words; each object has a linked list of all other objects referencing itself. The API has a special iterator to traverse this linked list, the *ReferencesIterator that is*.

In the example below one iterates over all product_definition_formation objects referring to a specific product.

```
// Create a second product_definition_formation object
  product_definition_formation* pdf2 = newObject(product_definition_formation);
  pdf2->put_description("Aircraft with two front doors and wheel landing gear");
  pdf2->put_id("2");
// Link product_definition_formation object to product object
  pdf2->put_of_product(p1);
  printf("\nVersion 2 for Product Aircraft 1 created.\n");

// Find product_definition_formation objects that are refering to product p1
  printf("\nFind product_definition_formation objects that are refering to product
p1");
  ReferencesIterator<product_definition_formation*, entityType>
pdfIter(p1,et_product_definition_formation);
  product_definition_formation *mypdf;
  for(mypdf = pdfIter.first(); mypdf; mypdf = pdfIter.next()){
    printf("\nVersion %s for product %s\n",mypdf->get_id(),mypdf->get_of_product()-
>get_id());
  }
```

If the program only needs one reference and/or there only exists one object of the specified type in the linked list, the *getFirstReferencing*(<wanted type>) can be used.
Example:

```
// Find first referencing of type product_definition_formation
   mypdf = (product_definition_formation*)p1-
>getFirstReferencing(et_product_definition_formation);
   printf("\nFirst referencing version for product %s is version %s\n",mypdf-
>get_of_product()->get_id(),mypdf->get_id());
```

If you want to handle all referencing objects regardless of role, the *AllReferencesIterator* is available. The *AllReferencesIterator* has two constructors. The first one with no parameters will return all objects when traversing. The second constructor accepts an array of object types, where the last value is et_indeterminate, This array specifies which object types are to be returned from *first*/*next* methods, a filter in other words.

Example:

```
// Create a product catgory and connect the products to it
   product_related_product_category *prpc = newObject(product_related_product_category);
   prpc->put_name("Product category 1");
   prpc->put_products_element(p1);
   prpc->put_products_element(p2);

// Find all referencing
   printf("\nFind all referencing product p1");
   entityType myType;
   AllReferencesIterator <entityType> productRefs(p1);
   for (void *obj = productRefs.first(&myType); obj; obj = productRefs.next(&myType)) {
     if(myType == et_product_definition_formation){
       product_definition_formation *pdf = (product_definition_formation*)obj;
       printf("\nReferencing : version %s for product %s",pdf->get_id(),pdf->get_of_product()-
>get_id());
     }
     if(myType == et_product_related_product_category){
       product_related_product_category *cat = (product_related_product_category*)obj;
       printf("\nCategory : %s",cat->get_name());
     }
   }
```

3.13 Store and retrieve objects in the database

The first use case for the *C++ EXPRESS API* was to implement converters that convert native CAD/CAE data to and from the ISO 10303-209 (AP209) standard. In this use case data are read from native CAD/CAE files into the memory and translated to an AP209 model in memory. Finally the objects in memory are written to an EDM model on the database in one operation. Two methods for database storage and retrieval are implemented.

1. Model.*writeAllObjectsToDatabase* – writes all objects in memory that belong to the Model object into the database model. The database model must be opened for write.
2. Model.*readAllObjectsToMemory* – reads all objects in an open database model into memory.

Before reading objects from a database model into memory, filters may be applied by means of the method *defineObjectSet*. The filter defined by *defineObjectSet* excludes all but one object type and its possible subtype.

```
// define an object set of product and all subtypes of product
myModel.defineObjectSet(et_product, 12, true);
myModel.readAllObjectsToMemory(&ma);

SdaiAggr productsList = (SdaiAggr)myModel.getObjectSet(et_product);
entityType productType, ct;

// create a iterator for traversing all products read into memory:
Model *myModelp = &myModel;
Iterator<product*, entityType> products(productsList, myModelp);
for (product *p = products.first(&productType); p; p = products.next(&productType))
{
    // handle the product
}
```

In case one wants to make use of a filter that includes several object types and their possible subtypes, the *readObjectClassesToMemory* method may be applied. It reads the objects into memory in one shot as well. It accepts an array of object class enumerations as input.

```
readObjectClassesToMemory(int *objectClasses)
```

Another variant of this method accepts an advanced filter. Its usage is explained in dbinterface.h .

3.14 Lack of memory

The *C++ EXPRESS API* creates objects in memory and in some applications number of objects can be so high that lack of memory becomes a problem. In such a case one should switch to the objects in database (OID) memory model. Applications must use the name space **OID**, and the pre-processor directive **USE_EDMI** must also be applied in this case.

3.15 Container in memory

cpp10 has a class for storing data in a list in memory. It is declared as follows:

```
template <class ContainerMember>
class Container : public ContainerImplementation
```

where ContainerMember is the type of the member.

Example: Using container to prevent that a name is printed several times from a stream where it can occur more than once.

```
bool haveNotBeenPrinted(Container<const char *> *printed, const char * name)
{
   if (! printed->contains(name)) {
      printed->add(name); return true;
   } else {
      return false;
   }
}

CmemoryAllocator *ma = ....
// Memory for all the Container members are allocated from ma
Container<const char *>    namesPrinted(&lma);

for all names int input stream {
   if (haveNotBeenPrinted(namesPrinted, name)) {
      printf("%s\n", name);
   }
}
```

See the reference documentation for details.

3.16 EXPRESS functionality not included in the CppExpressAPI

For the objects in memory (OIM) model, inverse and derived attributes are handled in the following way:

- Inverse attributes – when putting a value for an attribute that have an inverse, the inverse is not updated in client memory. But when the objects are written to the database and afterwards read into memory, the inverse attributes are put in place.
- Derived attributes – the implementation of derived attributes have a similar weakness as inverse attributes. They are assigned after they have been read into memory from the database. If the data that constitute a part of the foundation for the derived attribute

are later changed in memory, the derived attribute is not changed. Subsequently a new "roundtrip" to/from the database is required.

For both memory models:

- EXPRESS built in function as documented in chapter 15 of ISO-10303-11 are not included in the API.
- Rule validation is not included in the API.

These functionalities are, however, available if one applies the *EDMinterface* (C-API) directly. Functions available in the C-library (like TAN etc) are not replicated in the *EDMinterface.*

4. EDM Server Side C++ Plug-in

The EDM Server Side C++ Plug-ins are C++ programs that will be built as a dynamically linked libraries (.dll on Windows, .so on Linux). Such plugins will read and write to/from an EDM data model as if they were stand-alone programs that open the database exclusively. Such database access will be very efficient. Such plug-ins (.dll modules) shall be linked into the address space of the EDM application server on demand when the first call to a method in a plug-in .dll is executed.

An EDM plug-in has the following characteristics:

- It must implement the following 4 methods:
  - dll_main – Main entry of the plug-in. The first three parameters are string parameters that can be used by the plug-ins in different ways. A normal usage pattern for these parameters is to use them for repository, model and method name. Repository and model name is used to open the target model. The method name can be used as a switch within dll_main to execute the correct sub-method. Dll_main must also have other input parameters that are transferred from the client program to the plug-in, and return values that are transferred back to calling client program. Details will be explained later.
  - dll_version – Must return a 64 bits integer as the version number.
  - dll_malloc – method for allocating memory in the plug-in.
  - dll_freeAll – method called by EDM application server when all return values are sent back to the calling client program.
- If it shall access the EDM database, it must be linked with EDM database access library appserver600.lib.
- A plug-in method is invoked within an EDM application server by a client program executing the edmi method edmiRemoteExecuteCppMethod.
- Input parameters and return values shall be "packed" into an object with class name CppParameterClass for easy transfer between the client program and the EDM application server.

In the following we will describe how to implement a method that shall run as an EDM server side plug-in. Since input parameters are transferred from a client program to the EDM application server, and return values are transferred back to client program, we will explain in detail how this is done. To illustrate this functionality we make use of an example from the finite element analysis world. The server side method shall return info about meshes from an analysis done for one specific time steps – GetListOfMeshes. Input parameters to GetListOfMeshes are repositoryName, modelName, analysisID and timeStep. Return values are two strings and a list of records where each record holds information about one mesh. MeshInfo is defined below:

```
   struct MeshInfo {
      char                                 name[MAX_NAME_SIZE];
      EDMULONG                             n_vertices;
      EDMULONG                             n_elements;
   } MeshInfo;
```

To implement a call to the server method GetListOfMeshes one must perform the following steps:

Generic input parameters serverContextId , remoteRepositoryName, remoteModelName , pluginPath, pluginNameand methodName are the first six parameters of edmiRemoteExecuteCppMethod. The two other input parameters, analysisID and timeStep, are specific for the GetListOfMeshes method and must be transferred to the server within a parameter container InGetListOfMeshes, which is declared below.

Declare container for input parameters InGetListOfMeshes as subclass of CppParameterClass

```
class InGetListOfMeshes : public CppParameterClass
{
   cppRemoteParameter                     *attrPointerArr[2];
   cppRemoteParameter                     *analysisID;
   cppRemoteParameter                     *timeStep;
   void* operator new(size_t sz, CMemoryAllocator *ma){ return ma->alloc(sz); }
   InGetListOfMeshes(CMemoryAllocator *_ma, cppRemoteParameter *inAttrPointerArr)
       : CppParameterClass(attrPointerArr, sizeof(attrPointerArr), _ma,
       inAttrPointerArr)
   {
      addAddribute(&analysisID, rptSTRING);
      addAddribute(&timeStep, rptREAL);
   }
};
```

Each parameter must be declared as pointer attributes to the subclass of CppParameterClass. It is also required that a cppRemoteParameter pointer array with size equal to number of parameters. This pointer array is the pointer array that is used as parameter to edmiRemoteExecuteCppMethod

In the constructor addAttribute must be called for each parameter

```
      addAddribute(&analysisID, rptSTRING);
      addAddribute(&timeStep, rptREAL);
```

The second parameter to addAttribute is the type of the attribute. EDM supports the types of parameters listed in Table 2, below.

Table 2: EDMparameter types

Parameter types	Explanation
`rptINTEGER, rptREAL, rptBOOLEAN, rptLOGICAL, rptSTRING, rptENUMERATION`	EDM primitive types
`rptAggrINTEGER, rptAggrREAL,`	Array of 64 bits integer or real
`rptLongAggrINTEGER, rptLongAggrREAL,`	Same as above - the implementation use this to specify different handling.
`rptBLOB`	Continuous array of bytes
`rptContainer`	template <class ContainerMember> class Container Expandable container of any object. Explained in code examples below.
`rptStringContainer`	Special implementation of string container: Container<char *> myStrings = new.......
`rptCMemoryAllocator`	All memory buffers of the memory allocator are transferred as one parameter. There are methods to support relocation of pointers after transfer of the memory buffers to target process. Explained in code examples below.
`rptUndefined`	Empty / undefined parameter.

The return values container has the following return values attributes:

```
cppRemoteParameter                    *status;
cppRemoteParameter                    *report;
cppRemoteParameter                    *mesh_info_list;
```

and within the constructor the initialization is as follows:

```
addAddribute(&status, rptSTRING);
addAddribute(&report, rptSTRING);
addAddribute(&mesh_info_list, rptContainer);
```

4.1 Container as remote parameter

Input parameters to GetListOfMeshes are then declared as:

```
InGetListOfMeshes *inParams = new(ma)InGetListOfMeshes(ma, NULL);
```

And the input parameter values are set :

```
inParams->analysisID->putString((char*)analysisID.data());
inParams->timeStep->putReal(timeStep);
```

In the example a wrapper around edmiRemoteExecuteCppMethod, `ExecuteRemoteCppMethod` is used. Therefore will a call to GetListOfMeshes on the EDM application server look like:

```
RvGetListOfMeshes *retVal = new(ma)RvGetListOfMeshes(ma, NULL);
ExecuteRemoteCppMethod(repository, model, "GetListOfMeshes", inParams, retVal);
```

Within the C++ plug-in the dll_main method will look like the following:

```
extern "C" EDMLONG __declspec(dllexport) dll_main(
       char *repositoryName,
       char *modelName,
       char *methodName,
       EDMLONG nOfParameters,
       cppRemoteParameter *parameters,
       EDMLONG nOfReturnValues,
       cppRemoteParameter *returnValues,
       void **threadObject)
{
```

Below the code within the main switch within dll_main is shown. At his time in the execution the correct EDM model is opened, a memory allocator "ma" is created and the main plug-in object is ready to execute the GetListOfMeshes method. We also see that objects referring input parameters and return values are created. They are convenient wrappers around the actual parameters transferred from client program and buffers for return values.

```
if (strEQL(methodName, "GetListOfMeshes")) {
   RvGetListOfMeshes *results = new(ma)RvGetListOfMeshes(NULL, returnValues);
   InGetListOfMeshes *inParams = new(ma)InGetListOfMeshes(NULL, parameters);
   rstat = plugin->GetListOfMeshes(modelId, inParams, results);
```

The next step is the code that retrieves mesh information from the database and moves it to mesh info records in the mesh info container. The mesh info container is declared:

```
   Container<MeshInfo> *meshContainer = new(ma)Container<MeshInfo>(ma);
```

For all meshes within the specified analysis and time step within the specified EDM model the following is code applies. The fem::mesh pointer mesh is a pointer to the database object that holds information about the mesh and it is an attribute in time step database object.

```
       fem::Mesh *mesh = ts->get_mesh();
```

For each mesh a new mesh info record is created in the mesh info container by createNext().

```
        MeshInfo *mi = meshContainer->createNext();
```

Then the mesh information is moved into the mesh info record.

```
        strcpy(mi->name, mesh->get_name());
        mi->nElements = mesh->get_elements()->size();
        mi->nVertices = mesh->get_nodes()->size();
```

When all mesh info records are created the time has come for return to client program. Then set return values:

```
        retVal->mesh_info_list->putContainer(meshContainer);
        retVal->status->putString("OK");
        retVal->report->putString("");
```

When the client program get the return values from the server and the status return value is equal to "OK", it can create a container that takes its content from the buffers that are returned from the server.

```
    Container<MeshInfo> *meshInfos;

    meshInfos = new(ma)Container<MeshInfo>(ma, retVal->mesh_info_list);
```

Then the container can be handled as it was created locally. For example if the mesh information shall be written to file, it can look like this:

```
    MeshInfo *mi = meshInfos->firstp();
    while (mi) {
       mi = meshInfos->nextp();

       printf(filep, "%s, %Ul, %ul\n", mi->name, mi->n_elements, mi-> vertices);

    }
```

5. How to

5.1 Download C++ Express API and Express DataManager™

The C++ Express API is part of the *Express Data Manager*™ (EDM) installation. It can be downloaded from http://www2.epmtech.jotne.com/download/

5.2 Generate C++ files for EXPRESS schema

The C++ files representing an EXPRESS schema are generated by means of the *EDMsupervisor*™ command.

Schemata->Generate Interface->Cpp 2010

The command has the parameters and options listed in Table 3, below.

Table 3: Parameters and options for the Generate Cpp 2010 command

Parameter name	Description and options
Schema:	The name of the EXPRESS schema the generator shall generate C++ files for.
Cpp interface output directory:	The directory where the generated C++ files shall be written.
Namespace:	The C++ namespace used in the generated C++ files
Class addition folder:	It is possible to add declarations to the generated C++ classes. The additions must be written in a file located in the specified folder. The name of the file must be as follows: <entity name>_addition.h If you, for example, want to implement a method called getShape in the class fea_model, you write a file named fea_model_addition.h with the following content: public: product_definition_shape *getShape(); and store the file in the specified folder.
Complex entity file name:	In EXPRESS it is possible to compose entity definitions that are the union of other entities (object classes). These entities are called complex entities and the syntax is as follows: <entity_1>+<entity_2> The specified file shall contain one line for each complex entity the generator shall generate implementation for in addition to the entities in the specified schema.

Parameter name	Description and options
	A file containing the following line conversion_based_unit+mass_unit specifies that the generator shall generate a class named conversion_based_unit_mass_unit. The possible combinations are numerous and the *EDMcompiler* does not include any complex entities in the dictionary model. The dictionary model is updated on the fly the first time a proper complex entity is being instantiated. Such a pattern will not work for the generated C++ classes. One needs to anticipate up front which complex entities will be used in a project; but, if just one is forgotten, one must regenerate the C++ files after updating the complex entity file.

5.3 Use EDMinterface

The *EDMinterface*™ is a very comprehensive API, and it is neither practical nor necessary to include all its functionality in the *CppExpressAPI* (see chapter 3.16). But since *CppExpressAPI* is layered on top of *EDMinterface*, all its functionality is available. The prototypes of all functions are defined in **sdai.h**.
As mentioned in chapter 3.16, rule validation is not included in the *CppExpressAPI*, so we can use that as an example. There are various options for validation, but we can use the most common one, and exemplify a full validation of the whole population in a data model.

```
SdaiRepository rep = sdaiOpenRepositoryBN("DataRepository",sdaiRO);
EdmiError rstat = edmiValidateModelBN(rep,"MyModel","DiaFile",FULL_VALIDATION |
FULL_OUTPUT,NULL,NULL,validationErrorId);
```

Documentation of *EDMinterface* is found in **EDMAssist Volume IV**.
http://edmserver.epmtech.jotne.com/EDMassist/WebHelp/EDMAssist.htm

5.4 Build applications

The following include files from the include folder of the EDM installation, %EDM_HOME%\include, have to be included:

```
extern "C" {
#include "sdai.h"
#include "cpp10_EDM_interface.h"
}
#include "cpp10_container.h"
```

If you shall build an application using the "in memory" object option, you shall link with the following two libraries:

- *cpp10.lib* – the cpp10 library
- *edmikit600.lib* – EDM database interface

And in the application source you declare
using namespace OIM;

If you will build the "in database" object version you compile with preprocessor directive USE_EDMI set and link with the following libraries

- *cpp10_edmi.lib*
- *edmikit600.lib*

And in the application source declare
using namespace OID;

For both memory models compile and build with 64bits option.

Appendix A – OIM example

This example makes use of the **Objects In Memory** pattern.
The example is based on the AP209 schema and to prepare for running the example, one must perform the following use the *EDMSupervisor*:

- Create a database by means
  Database->Create
- Compile the AP209 schema.
  Schemata->DefineSchema
- Generate C++ classes for the AP209 schema (files to be included in the VS project)
  Schemata->GenerateInterface->Cpp2010 (specify namespace ap209)
- Create a model (with name myModel)
  Data->Create->Model
- Close the database
  Database->Close

Contents of the **stdafx.h** file:

```
#pragma once
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
extern "C" {
#include "sdai.h"
#include "cpp10_EDM_interface.h"
}
#include "cpp10_container.h"
using namespace OIM;
#include
"ap209_multidisciplinary_analysis_and_design_mim_lf_entityTypes.h"
#include "ap209_multidisciplinary_analysis_and_design_mim_lf.hpp"
using namespace ap209;
```

```
// TestCppexpressApi_OIM.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#define newObject(className) new(m)className(m)
int _tmain(int argc, _TCHAR* argv[])
{

   try{
     EdmiError     rstat;
     SdaiInteger   nbWarnings;
     SdaiInteger   nbErrors;
     SdaiErrorCode sdaiError;

     Database db("C:/edm/EDMinterface AP209-CPP/db","d","d");
     db.open();
     Repository cRepository(&db, "DataRepository");

     ap209_Schema my_Schema = ap209_multidisciplinary_analysis_and_design_mim_lf_SchemaObject;
     CMemoryAllocator ma(1000000);
     Model myModel(&cRepository, &ma, &my_Schema);
     myModel.open("myModel", sdaiRW);
     Model *m = &myModel;

     printf("\nMEMORY_MODEL : objects in memory.\n");
     // Create product object and set its attributes
     product* p1 = newObject(product);
     p1->put_id("Aircraft 1");
     p1->put_name("Aircraft propeller map");
     p1->put_description("Aircraft 1");
     printf("\nProduct Aircraft 1 created.");

     // Create product_definition_formation object
     product_definition_formation* pdf = newObject(product_definition_formation);
     pdf->put_description("Aircraft with two front doors and wheel landing gear");
     pdf->put_id("1");
     // Link product_definition_formation object to product object
     pdf->put_of_product(p1);
```

```
printf("\nVersion 1 for Product Aircraft 1 created.");

// Create product_definition object
product_definition* pd = newObject(product_definition);
pd->put_description("description");
// Link product_definition object to
// product_definition_formation object
pd->put_formation(pdf);
pd->put_id("pd-1");
printf("\nProduct definition pd-1 for Version 1 created.");

// Create a property
property_definition *prop_def_1= newObject(property_definition);
prop_def_1->put_name("Prop1");
// set the definition attribute beeing the product definition from above
prop_def_1->put_definition(pd);

// Retrieve the proprty definition and exemplify how is processed
// based on the retrieved type.
entityType objectType;
void* obj = prop_def_1->get_definition(&objectType);
if(objectType == et_product_definition){
  product_definition *mypd = (product_definition*)obj;
  product *myp = mypd->get_formation()->get_of_product();
  printf("\nProperty 1 is connected to product: %s",myp->get_id());
} else if(objectType == et_characterized_object){
// handle characterized_object
} else if(objectType == et_product_definition_shape){
// handle product_definition_shape
} else if(objectType == et_shape_aspect){
// handle shape_aspect
} else if(objectType == et_shape_aspect_relationship){
// handle shape_aspect_relationship
} else if(objectType == et_product_definition_relationship){
// handle product_definition_relationship
} else {
  throw new CedmError(sdaiETYPEUNDEF);
```

```
    }

    // Create another property and attach it to a characterized_object
    property_definition *prop_def_2= newObject(property_definition);
    prop_def_2->put_name("Prop2");
    characterized_object *co = newObject(characterized_object);
    co->put_name("co1");
    prop_def_2->put_definition(co);

    // Retrieve the proprty definition and exemplify how is processed
    // based on the retrieved type.     obj = prop_def_2->get_definition(&objectType);
    if(objectType == et_product_definition){
    // handle product_definition
    } else if(objectType == et_characterized_object){
      characterized_object *myco = (characterized_object*)obj;
      printf("\nProperty 2 is connected to characterized_object: %s",myco->get_name());
    } else if(objectType == et_product_definition_shape){
    // handle product_definition_shape
    } else if(objectType == et_shape_aspect){
    // handle shape_aspect
    } else if(objectType == et_shape_aspect_relationship){
    // handle shape_aspect_relationship
    } else if(objectType == et_product_definition_relationship){
    // handle product_definition_relationship
    }else {
      throw new CedmError(sdaiETYPEUNDEF);
    }

   // The code below shows how to create curve_element_freedom selects
   // and put then into the attribute release_freedom of  //_curve_element_end_release_packet objects
   // By new(m), the selects are created in memory controlled Models
   // The m after enumerated_curve_element_freedom_X_ROTATION is necessary
   // beacause enumerated_curve_element_freedom_X_ROTATION must be translated
   // to the corresponding instance identifier in the underlying database.
    curve_element_freedom  *cef1;
    cef1 = new(m)curve_element_freedom(enumerated_curve_element_freedom_X_ROTATION, m);
    curve_element_freedom  *cef2;
```

```
cef2 = new(m)curve_element_freedom("application defined degree of fredom",m);

curve_element_end_release_packet * ceerp1;
ceerp1 = newObject(curve_element_end_release_packet);
ceerp1->put_release_freedom(cef1);
curve_element_end_release_packet * ceerp2;
ceerp2 = newObject(curve_element_end_release_packet);
ceerp2->put_release_freedom(cef2);

// SET example
// First create another product
product* p2 = newObject(product);
p2->put_id("Aircraft 2");
p2->put_name("Aircraft propeller map");
p2->put_description("Aircraft 2");
printf("\nProduct Aircraft 2 created.");

// Allocate SET OF products
Set<action_items*>* prodSet = new(&ma) Set<action_items*>(&ma, sdaiINSTANCE, 2);
// Add the two products already created
prodSet->add(p1,&ma);
prodSet->add(p2,&ma);
// Assign set to attribute items of applied_action_assignment
applied_action_assignment *aaa = newObject(applied_action_assignment);
aaa->put_items(prodSet);

// ITERATORS :
// Iterator for the set of products
printf("\n\nProducts in set:");
Iterator<product*, entityType> prodIter(prodSet);
int i =0;
for(product *prod = prodIter.first(); prod; prod = prodIter.next()){
  ++i;
  printf("\nElement : %d - %s",i,prod->get_id());
}
// References iterator
// Create a second product_definition_formation object
```

```
product_definition_formation* pdf2 = newObject(product_definition_formation);
pdf2->put_description("Aircraft with two front doors and wheel landing gear");
pdf2->put_id("2");
// Link product_definition_formation object to product object
pdf2->put_of_product(p1);
printf("\nVersion 2 for Product Aircraft 1 created.\n");

// Find product_definition_formation objects that are refering to product p1
printf("\nFind product_definition_formation objects that are refering to product p1");
ReferencesIterator<product_definition_formation*, entityType> pdfIter(p1,et_product_definition_formation);
product_definition_formation *mypdf;
for(mypdf = pdfIter.first(); mypdf; mypdf = pdfIter.next()){
  printf("\nVersion %s for product %s",mypdf->get_id(),mypdf->get_of_product()->get_id());
}

// Find first referencing of type product_definition_formation
mypdf = (product_definition_formation*)p1->getFirstReferencing(et_product_definition_formation);
printf("\nFirst referencing version for product %s is version %s\n",mypdf->get_of_product()->get_id(),mypdf->get_id());

// AllReferencesIterator iterator
// Create a product catgory and connect the products to it
product_related_product_category *prpc = newObject(product_related_product_category);
prpc->put_name("Product category 1");
prpc->put_products_element(p1);
prpc->put_products_element(p2);

// Find all referencing
printf("\nFind all referencing product p1");
entityType myType;
AllReferencesIterator <entityType> productRefs(p1);
for (void *obj = productRefs.first(&myType); obj; obj = productRefs.next(&myType)) {
  if(myType == et_product_definition_formation){
    product_definition_formation *pdf = (product_definition_formation*)obj;
    printf("\nReferencing : version %s for product %s",pdf->get_id(),pdf->get_of_product()->get_id());
  }
  if(myType == et_product_related_product_category){
    product_related_product_category *cat = (product_related_product_category*)obj;
```

```
        printf("\nCategory : %s",cat->get_name());
      }
    }

    // LIST example
    List<STRING> theWeekDays(&ma, sdaiSTRING, 7);
    theWeekDays.add("Sunday", &ma);
    theWeekDays.add("Monday", &ma);
    theWeekDays.add("Tuesday", &ma);
    theWeekDays.add("Wednesday", &ma);

    // ARRAY Examples
    // Declare array of int with min index 5 and max index 15
    Array<EDMLONG> arr(&ma, sdaiINTEGER, 5, 15);
    // Initialize the array
    for (EDMLONG i=5; i <= 15; i++) arr[i] = 100 + i;
    // Print the array values
    printf("\n\nInteger array :\n");
    for (EDMLONG i=5; i <= 15; i++) {
      printf("arr[%d] = %d\n", i, arr[i]);
    }

    // An example using ap209 euler_angles
    parametric_volume_3d_element_coordinate_system *pv3d = newObject(parametric_volume_3d_element_coordinate_system);
    Array<double>angles (&ma, sdaiREAL, 1, 3);
    angles[1] = 2.71; angles[2] = 3.14; angles[3] = 6.58;
    euler_angles *euang = newObject(euler_angles);
    euang->put_angles(&angles);
    pv3d->put_eu_angles(euang);

    // The entity euler_angles have an array that is retrieved
    // by the method ea->get_angles();
    euler_angles *ea = pv3d->get_eu_angles();
    Array<double> *myangles = ea->get_angles();
    double angle1 = myangles->get(1);
    double angle2 = myangles->get(2);
    double angle3 = myangles->get(3);
```

```
      printf("\n\nEuler angles : %f,%f,%f",angle1,angle2,angle3);

      // Write objects in memory to database
      m->writeAllObjectsToDatabase();
      // Export the resulting population to step file.
      rstat = edmiWriteStepFile("DataRepository",NULL,"myModel","C:/Temp/myStepFile.stp",NULL,NULL,0,8,&nbWarnings,&nbErrors,&sdaiError);
       // Apply the next statement if you want to run more than one time.
      //rstat = edmiDeleteModelContentsBN("DataRepository","myModel");
      myModel.close();
    } catch (CedmError *e) {
       if (e->message) {
          printf(e->message);
       } else {
          printf(edmiGetErrorText(e->rstat));
       }
    } catch (int thrownRstat) {
       printf(edmiGetErrorText(thrownRstat));
    }

    printf("\n\nPress return to exit.");
    char c = getchar();
    return 0;
 }
```

Appendix B – OID example

This example makes use of the **Objects In Database** pattern.
Apply the same steps as in Appendix A to prepare the database.

Contents of the **stdafx.h** file:

```
#pragma once
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
extern "C" {
#include "sdai.h"
#include "cpp10_EDM_interface.h"
}
#include "cpp10_container.h"
using namespace OID;
#include "ap209_multidisciplinary_analysis_and_design_mim_lf_entityTypes.h"
#include "ap209_multidisciplinary_analysis_and_design_mim_lf.hpp"
using namespace ap209;
```

```
// TestCppexpressApi_OID.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
   try{
     EdmiError     rstat;
     SdaiInteger   nbWarnings;
     SdaiInteger   nbErrors;
     SdaiErrorCode sdaiError;

     Database db("C:/edm/EDMinterface AP209-CPP/db","d","d");
     db.open();
     Repository cRepository(&db, "DataRepository");

     ap209_Schema my_Schema = ap209_multidisciplinary_analysis_and_design_mim_lf_SchemaObject;
     CMemoryAllocator ma(1000000);
     Model myModel(&cRepository, &ma, &my_Schema);
     myModel.open("myModel", sdaiRW);
     Model *m = &myModel;

       printf("\nMEMORY_MODEL : objects in database.\n");
       product p1(m);
       p1.put_id("Aircraft 1");
       p1.put_name("Aircraft propeller map");
       p1.put_description("Aircraft 1");
       printf("\nProduct Aircraft 1 created.");

     // Create product_definition_formation object
       product_definition_formation pdf(m);
       pdf.put_description("Aircraft with two front doors and wheel landing gear");
       pdf.put_id("1");
     // Link product_definition_formation object to product object
       printf("\nVersion 1 for Product Aircraft 1 created.");
       pdf.put_of_product(&p1);
```

```
// Create product_definition object
  product_definition pd(m);
  pd.put_description("description");
// Link product_definition object to
// product_definition_formation object
  pd.put_id("pd-1");
  pd.put_formation(&pdf);
  printf("\nProduct definition pd-1 for Version 1 created.");

// Create a property
property_definition prop_def_1(m);
prop_def_1.put_name("Prop1");
// set the definition attribute beeing the product definition from above
prop_def_1.put_definition(&pd);

// Retrieve the proprty definition and exemplify how is processed
// based on the retrieved type.
entityType objectType;
void* obj = prop_def_1.get_definition(&objectType);
if(objectType == et_product_definition){
  product_definition *mypd = (product_definition*)obj;
  product *myp = mypd->get_formation()->get_of_product();
  printf("\nProperty 1 is connected to product: %s",myp->get_id());
} else if(objectType == et_characterized_object){
// handle characterized_object
} else if(objectType == et_product_definition_shape){
// handle product_definition_shape
} else if(objectType == et_shape_aspect){
// handle shape_aspect
} else if(objectType == et_shape_aspect_relationship){
// handle shape_aspect_relationship
} else if(objectType == et_product_definition_relationship){
// handle product_definition_relationship
} else {
  throw new CedmError(sdaiETYPEUNDEF);
}
```

```
// Create another property and attach it to a characterized_object
property_definition prop_def_2(m);
prop_def_2.put_name("Prop2");
characterized_object co(m);
co.put_name("co1");
prop_def_2.put_definition(&co);

// Retrieve the proprty definition and exemplify how is processed
// based on the retrieved type.
obj = prop_def_2.get_definition(&objectType);
if(objectType == et_product_definition){
// handle product_definition
} else if(objectType == et_characterized_object){
  characterized_object *myco = (characterized_object*)obj;
  printf("\nProperty 2 is connected to characterized_object: %s",myco->get_name());
} else if(objectType == et_product_definition_shape){
// handle product_definition_shape
} else if(objectType == et_shape_aspect){
// handle shape_aspect
} else if(objectType == et_shape_aspect_relationship){
// handle shape_aspect_relationship
} else if(objectType == et_product_definition_relationship){
// handle product_definition_relationship
}else {
  throw new CedmError(sdaiETYPEUNDEF);
}

// The code below shows how to create curve_element_freedom selects
// and put then into the attribute release_freedom of  //_curve_element_end_release_packet objects
// By new(m), the selects are created in memory controlled Models
// The m after enumerated_curve_element_freedom_X_ROTATION is necessary
// beacause enumerated_curve_element_freedom_X_ROTATION must be translated
// to the corresponding instance identifier in the underlying database.
curve_element_freedom  *cef1;
cef1 = new(m)curve_element_freedom(enumerated_curve_element_freedom_X_ROTATION, m);
curve_element_freedom  *cef2;
cef2 = new(m)curve_element_freedom("application defined degree of fredom",m);
```

```
curve_element_end_release_packet ceerp1(m);
ceerp1.put_release_freedom(cef1);
curve_element_end_release_packet ceerp2(m);
ceerp2.put_release_freedom(cef2);

// SET example
// First create another product
product p2(m);
p2.put_id("Aircraft 2");
p2.put_name("Aircraft propeller map");
p2.put_description("Aircraft 2");
printf("\nProduct Aircraft 2 created.");

// Allocate SET OF products
Set<action_items*>* prodSet = new(&ma) Set<action_items*>(m, sdaiINSTANCE, 2);
// Assign set to attribute items of applied_action_assignment
applied_action_assignment aaa(m);
aaa.put_items(prodSet);
// Add the two products already created
prodSet->add(&p1);
prodSet->add(&p2);

// ITERATORS
// Iterator for the set of products
printf("\n\nProducts in set:");
product *prod1 = NULL;
product *prod2 = NULL;
entityType mytype;
int i = 0;
Iterator<product*, entityType> prodIter(aaa.get_items());
for(product *prod = prodIter.first(); prod; prod = prodIter.next()){
  printf("\nElement : %s",prod->get_id());
  ++i;
  if(i ==1)
    // A clone that is present after the loop
    prod1 = (product*)m->clone(prod,(int*)&mytype);
  if(i ==2)
    // A clone that is present after the loop
    prod2 = (product*)m->clone(prod,(int*)&mytype);
}
printf("\nProduct 1 after iteration: %s",prod1->get_id());
printf("\nProduct 2 after iteration: %s",prod2->get_id());

// ReferencesIterator
// Create a second product_definition_formation object
product_definition_formation pdf2(m);
pdf2.put_description("Aircraft with two front doors and wheel landing gear");
pdf2.put_id("2");
// Link product_definition_formation object to product object
pdf2.put_of_product(&p1);
printf("\nVersion 2 for Product Aircraft 1 created.\n");

// Find product_definition_formation objects that are refering to product p1
printf("\nFind product_definition_formation objects that are refering to product p1");
ReferencesIterator<product_definition_formation*, entityType> pdfIter(&p1,et_product_definition_formation);
product_definition_formation *mypdf;
for(mypdf = pdfIter.first(); mypdf; mypdf = pdfIter.next()){
  printf("\nVersion %s for product %s",mypdf->get_id(),mypdf->get_of_product()->get_id());
}

// Find first referencing of type product_definition_formation
mypdf = (product_definition_formation*)p1.getFirstReferencing(et_product_definition_formation);
printf("\nFirst referencing version for product %s is version %s\n",mypdf->get_of_product()->get_id(),mypdf->get_id());

// AllReferencesIterator iterator
// Create a product catgory and connect the products to it
product_related_product_category prpc(m);
prpc.put_name("Product category 1");
prpc.put_products_element(&p1);
prpc.put_products_element(&p2);

// Find all referencing
printf("\nFind all referencing product p1");
```

```
entityType myType;
AllReferencesIterator <entityType> productRefs(&p1);
for (void *obj = productRefs.first(&myType); obj; obj = productRefs.next(&myType)) {
  if(myType == et_product_definition_formation){
    product_definition_formation *pdf = (product_definition_formation*)obj;
    printf("\nReferencing : version %s for product %s",pdf->get_id(),pdf->get_of_product()->get_id());
  }
  if(myType == et_product_related_product_category){
    product_related_product_category *cat = (product_related_product_category*)obj;
    printf("\nCategory : %s",cat->get_name());
  }
}

// ARRAY Examples
// An example using ap209 euler_angles
parametric_volume_3d_element_coordinate_system pv3d(m);
Array<double>angles (m, sdaiREAL, 1, 3);
euler_angles euang(m);
euang.put_angles(&angles);
pv3d.put_eu_angles(&euang);
// Please note that the assignment of values to angles must occur after array has been
// assigned to an instance already created in the database.
angles.put(1,2.71);
angles.put(2,3.14);
angles.put(3,6.58);

// The entity euler_angles have an array that is retrieved
// by the method ea->get_angles();
euler_angles *ea = pv3d.get_eu_angles();
Array<double> *myangles = ea->get_angles();
double angle1 = myangles->get(1);
double angle2 = myangles->get(2);
double angle3 = myangles->get(3);
printf("\n\nEuler angles : %f,%f,%f",angle1,angle2,angle3);

// INHERITANCE example :
// First create a representation_context and three objects that all are subtypes of
// representation namely a node, fea_model_3d and explicit_element_representation
printf("\n\nInheritance:");
representation_context rep_ctx(m);
rep_ctx.put_context_identifier("Context1");
node n(m);
n.put_name("Node 1");
n.put_context_of_items(&rep_ctx);

fea_model_3d feam(m);
feam.put_name("Feam 1");
feam.put_creating_software("Lorentz developed the software");
feam.put_context_of_items(&rep_ctx);

explicit_element_matrix elm(m);
matrix_property_type mpt;
mpt.setString("Property type");
elm.put_property_type(&mpt);
explicit_element_representation eep(m);
eep.put_name("explicit_element_representation 1");
eep.put_matrix(&elm);
eep.put_context_of_items(&rep_ctx);

// Then get the representations attached to the context
// and handle them separately
Set<representation*> *reps;
reps = rep_ctx.get_representations_in_context();
entityType repType;
Iterator<representation*, entityType> repIter(reps);
for (representation* rep = repIter.first(&repType); rep; rep = repIter.next(&repType)) {
  printf("\nObject name : %s",rep->get_name());
  if (repType == et_node) {
    node *mynode = (node*)rep;
    printf("\nNode : %s",rep->get_name());
  }
  if (repType == et_fea_model_3d) {
    fea_model_3d *myfeam = (fea_model_3d*)rep;
    printf("\nfea_model_3d : %s",rep->get_name());
```

```
          printf("\nfea_model_3d software: %s",myfeam->get_creating_software());
        }
        if (repType == et_explicit_element_representation) {
          explicit_element_representation *myeet = (explicit_element_representation*)rep;
          printf("\nexplicit_element_representation : %s",rep->get_name());
          explicit_element_matrix *myeem = myeet->get_matrix();
          matrix_property_type *mympt = myeem->get_property_type();
          char *mychar = mympt->getString();
          printf("\nMatrix property type : %s",mychar);
        }
      }

      // Export the resulting population to step file.
      rstat = edmiWriteStepFile("DataRepository",NULL,"myModel","C:/Temp/myStepFile.stp",NULL,NULL,0,8,&nbWarnings,&nbErrors,&sdaiError);
      // Apply the next statement if you want to run more than one time.
      //rstat = edmiDeleteModelContentsBN("DataRepository","myModel");
      myModel.close();
    } catch (CedmError *e) {
       if (e->message) {
          printf("\n%s",e->message);
       } else {
          printf("\n%s",edmiGetErrorText(e->rstat));
       }
    } catch (int thrownRstat) {
       printf("\n%s",edmiGetErrorText(thrownRstat));
    }

    printf("\n\nPress return to exit.");
    char c = getchar();
    return 0;
}
```