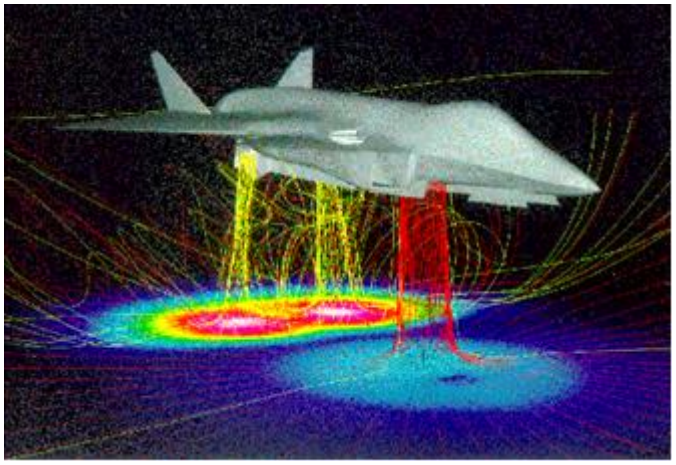
Open Simulation Data Management (Open SimDM)



AP209ed2 API User's Guide

CONFIDENTIALITY

Prepared by:
Jotne EPM Technology, Oslo, Norway
Grenseveien 107, P.O.Box 6629 Etterstad,
NO-0607 Oslo, Norway
Phone: +47 23 17 17 00
Fax: +47 23 17 17 01
E-mail: olav.liestol@jotne.com

TABLE OF CONTENT

LIST OF FIGURES

ABBREVIATIONS

AIM	Application Interpreted Model
ARM	Application Reference Model
AP209	ISO/CD 10303-209e2
API	Application Programming Interface
ATS	Abstract Test Suite
BOM	Business Object Model
CAD	Computer-Aided Design
CAE	Computer-Aided Engineering
CFD	Computational Fluid Dynamics
EDM	EXPRESS Data Manager™
EXPRESS	Data modeling language, defined in ISO 10303-11
ISO	International Organization for Standardization
ISO 10303	Industrial automation systems and integration — Product data representation and exchange
ISO 10303-11	Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual
ISO 10303-21	Industrial automation systems and integration — Product data representation and exchange: Implementation methods: Clear text encoding of the exchange structure
ISO 10303-26	Industrial automation systems and integration — Product data representation and exchange: Implementation methods: Binary representation of EXPRESS-driven data
ISO 10303-209	Industrial automation systems and integration — Product data representation and exchange — Part 209: Application protocol: Multidisciplinary Analysis and Design; in this context this is the CD-version of edition 2
Jotne	Jotne Group, including Jotne EPM Technology AS and Jotne North America Inc.
MIM	Module Interpreted Model
NASTRAN	Widely used Finite Element Analysis (FEA) solver
PDF	Portable Document Format
STEP	Standard for the Exchange of Product Model Data

REVISION HISTORY

Rev. no.	Date	Initials	Comment
1.0	2011-05-20	OLI	Initial version
1.1	2012-06-21	OLI	Update for pilots
1.2	2012-27-11	TJT	Edited for clarity
1.3	2012-12-13	OLI	Minor updates

1 Scope of the document

This document describes the AP209e2 Application Programming Interface (API) of the SimDM application and how to use the AP209 API when interfacing with other AP209 applications. The current version of the API is tailored towards implementing translators from a native data model to the ISO 10303-209e2 model (AP209e2).

The AP209 API uses the AP209 Business Object Model (AP209 BOM) because:

1. It is a layer that simplifies access to the ISO 10303 AP209e2 MIM (Module Implementation Model)
2. Ease of implementation and improve implementation performance.
3. AP209 BOM is a domain-specific model that is documented in the jargon most familiar to the target audience. This makes it more accessible to readers who are not familiar with the complex formal data model ISO 10303 AP209 MIM.

Not all the information requirement for AP209 are satisfied by business objects. Therefore the AP209 BOM model is a set of business objects supplied with AP209e2 MIM objects as required. In the AP209 BOM schema, the names of the business objects (BOs) have "BO_" as a prefix. The BOs may have AP209e2 MIM objects as attributes.

To store an AP209 BOM dataset in ISO 10303-209e2 format, the business objects must be translated into ISO 10303-209e2 MIM. In the current AP209 API this task is performed by the AP209objectFactory object. The AP209objectFactory also, as the name indicates, creates business objects. The translation to MIM objects is done at object creation time or when attributes later are changed. The AP209objectFactory has a method for creating each of the BOs. The methods have the name syntax new_<business object name>.

The AP209 API is implemented by the EDM C++ Express API which is an early binding interface to the Jotne EXPRESS Data Manager (EDM) model with an arbitrary EXPRESS schema. The necessary C++ files for using this API is generated by a command in the *EDMsupervisor*™

1.1 Document structure

This document is structured in the following manner:

- In this chapter the scope and structure of this document is presented.
- Chapter 2 enumerates other material and documents that are referenced.
- Chapter 3 is the user's guide to the generation of C++ APIs for EXPRESS models.
- Chapter 4 describes the Business Object API that is built on top of the C++ API.
- Chapter 5 contains the mapping from the Business Object API to AP209e2 AIM entities and attributes.
- Chapter 6 provides tutorials that are organized in such a way that it takes the reader through two examples. The main intention is to support development of translators that convert from native representation to AP209 populations. Hence the focus is on constructing business objects and saving them to the

database. Reading objects from the database for further processing is not demonstrated. The second tutorial example gives detailed information of how to compile and link programs using the AP209 API.

- Appendix A lists and explains the parameters for the automatic generation of parts of the API.

2 References

[1] ISO 10303-209, Industrial automation systems and integration — Product data representation and exchange — Part 209: Application protocol: Composite and metallic structural analysis and related design

[2] ISO/CD 10303-209e2, Industrial automation systems and integration — Product data representation and exchange — Part 209: Application protocol: Multidisciplinary analysis and design

[3] ISO 10303-21, Industrial automation systems and integration — Product data representation and exchange — Part 21: Implementation methods: Clear text encoding of the exchange structure

[4] ISO 10303-26, Industrial automation systems and integration — Product data representation and exchange — Part 26: Implementation methods: Binary representation of EXPRESS-driven data

[5] EDMassist VOLUME IV: *EDMinterface*™ Application Development Guide and Binding Reference

[6] Open SimDM Application - Reference Manual

3 C++ Express API

The main requirements of the EDM C++ Express API are:

- Client C++ library for handling Express data stored in a database on a database server.
- High performance. This is achieved by 1) minimizing the number of messages sent back and forth between the client and the server machine and 2) minimizing the volume of the data transferred. This is achieved by implementing a native protocol without textual overhead. SOAP is an example of a protocol that has textual overhead..
- C++ early binding.

A project using the C++ Express API starts with generating C++ header (.hpp) and implementation (.cpp) files from the Express schema that shall be used in the project. *EDMsupervisor*™ has the command Schemata->Generate Interface->Cpp 2010 for this. The header file will contain one C++ class for each entity and one type definition for each type in the Express schema.The different types in an Express schema are mapped to C++ in the following way:

Express	**C++**
Entity	Class The attributes are accessed via put and get methods.
Complex entity	Class with name equal to complex entity name except that the '+' delimiter is exchanged 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 to all C++ classes representing entities. Otherwise the select is mapped to cppSelect which is identical to the select struct used in SDAI. The next version of the generator will generate a more sophisticated representation of selects with put and get methods for all alternatives and methods for checking which of the alternatives it represents.
Enumeration	typedef enum {.., .., } <Enumeration name>;
Integer	int
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, in addition to constructors, have put and get methods for each explicit attribute. Derived and inverse attributes have only get methods.

Every C++ class that represents an Express entity inherits the class dbInstance. The dbInstance class contains reference 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.

In the C++ Express library it is possible to link objects together forming big structures in the client memory without any connection with the database server. When the structure is finished, it is possible to store the entire structure in the database with one single call. The C++ Express library at the client side will then serialize all objects and send them to the server. Used-in links are established automatically from the referred object to the referring object when the referring object set up direct link to the referred object.

3.1 Memory management in client process

The C++ Express API is designed to handle large amounts of data very efficiently. To meet this requirement, memory for the C++ objects on client side 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 collectively by deleting the CMemoryAllocator object or by executing the ma->Reset() method. The benefit of this way of deleting objects is better performance and almost no probability for memory fragmentation.

3.2 Database, Repository, Model, Schema and EDMI

A dataset 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 [5] - EDMassist VOLUME IV: *EDMinterface*™ Application Development Guide and Binding Reference.

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 call to EDMI
db.open();
// Repositor is declared
Repository cRepository(&db, "repository_name");

// myModel is declared with reference to memory allocator and schema.
// The memory allocator gives memory to the objects in the model.
// my_Schema is a reference to the generated C++ representation
// of the Express schema of myModel.
Model myModel(&cRepository, &ma, &my_Schema);
// nyModel is opened for read and write
myModel.open("myModel", sdaiRW);
```

3.3 Working with objects

3.3.1 Create objects in memory

In the C++ Express API objects are created with a reimplementation of the *new* operator, which replaces the default C++ *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.

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 part. The following C++ code sequence shows 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)
```

```
CMemoryAllocator *map;
Model *m;

// Create product object and set its attributes
product* p = newObject(product);
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 =
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(p);

// 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);
```

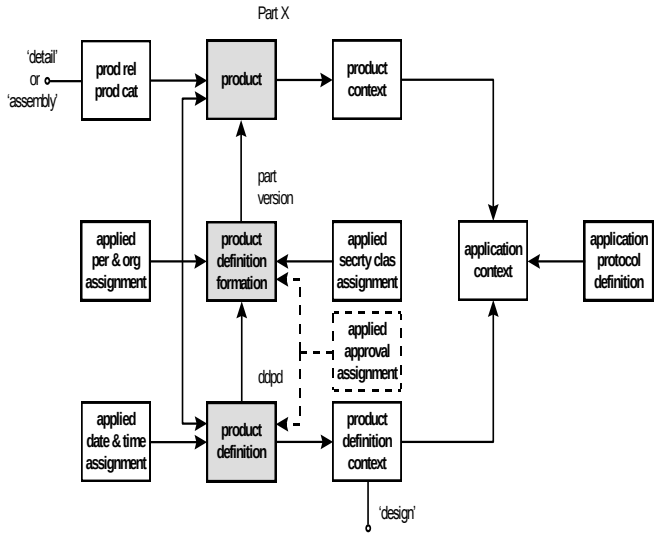


Figure 1 - High Level AP 209 Requirements for Parts

3.3.2 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;
```

3.3.3 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, 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 struct cppSelect, which is the same way as selects are treated in SDAI.

3.3.3.1 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 may be used where such selects are specified. When such selects are return values in for example get functions, the generated C++ type of the return value is void *. The void * can 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;
/*
if (objectType == et_characterized_object) {
   characterized_object *p = (characterized_object*)obj;
} else if (objectType == et_characterized_product_definition) {
   characterized_product_definition *p =
    (characterized_product_definition*)obj;
} else if (objectType == et_shape_definition) {
   shape_definition *p = (shape_definition*)obj;
} */
```

The characterized_definition is defined as dbInstance, the common supertype of all objects. The comment below the definition is meant as a help for handling the alternatives situations that might occur.

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

```
property_definition *pd = newObject(property_definition );
shape_definition *sd = newObject(shape_definition);

// set the definition attribute
pd->put_defintion(sd);

// get the definition attribute without knowing its exact entitytype
entityType defType = pd->typeof_definition();
if (defType == et_characterized_object) {
    characterized_object *co =
    (characterized_object*)pd->get_defintion();
    // implement handling of characterized_object
} else if (defType == et_characterized_product_definition) {
    characterized_product_definition *cpd =
    (characterized_product_definition*)pd->get_defintion();
    // implement handling of characterized_product_definition
} else if (defType == et_shape_definition) {
    shape_definition *sd = (shape_definition*)pd->get_defintion();
    // implement handling of shape_definition
} else {
    throw error;
}

// as an alternative to pd->typeof_definition(); and
// pd->get_defintion();
// void * get_definition(entityType *etp);
// is generated
```

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

See the Express example below. There 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;

// Select is created
// ma is memory allocator object
curve_element_freedom  *cef = new(ma)curve_element_freedom();
enumerated_curve_element_freedom cefValue;
cefValue = enumerated_curve_element_freedom_X_ROTATION;
// The select becomes an enumeration
cef->put_enumerated_curve_element_freedom(cefValue);
// The select becomes a string
cef->put_application_defined_degree_of_freedom("my_degree");
// There are bool methods returning true if the select is the
// alternative mentioned in the method name
bool isString = cef->is_application_defined_degree_of_freedom();
bool isEnum = cef->is_enumerated_curve_element_freedom();
// Get methods
cefValue = cef->get_enumerated_curve_element_freedom();
char *cefString = cef->get_application_defined_degree_of_freedom();
```

3.3.4 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

Two List examples:

```
// define memoryallocator
CMemoryAllocator ma(1000000);

// Allocate LIST OF Ply
List<Ply*>* plyList = new(&ma) List<Ply*>(&ma, sdaiINSTANCE, 2);
// Allocate a Length_data_element and Unit object
Length_data_element* lde = newObject(Length_data_element);
Unit* meter = newObject(Unit);
meter->put_name("meter");
// Add Unit with name "meter" to Length_data_element
lde->put_unit(meter, &ma);
// create 2 Ply objects
Ply* ply1 = newObject(Ply);
ply1->put_ply_thickness(lde, &ma);
Ply* ply2 = newObject(Ply);
// add the two Ply objects to the plyList
plyList->add(ply1, &ma);
plyList->add(ply2, &ma);


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

3.3.4.1 Put element methods

```
class BO_Processed_core
{
public:
   List<BO_Ply*>*       get_added_material();
   void                 put_added_material(List<BO_Ply*>* v);
   void                 put_added_material_element(BO_Ply*);
};
```

3.3.5 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 the aggregate. The aggregate is input parameter to the Iterator constructor.

An Iterator has a first() method that returns the first element in the aggregate. For all three aggregate types, Bag, Set or List, this is the first element added to the aggregate that is not removed.

After getting the first element of the aggregate it is possible to get the next element of the aggregate by the next() method.

After getting an aggregate element by either the first() or the next() method, one must check if the returned value represent an element. For pointers elements like object, aggregate or select pointer one simply check if the pointer is different from NULL. For elements that is not pointer like Enumeration, Integer, Real, Logical or Boolean one must use the method moreElems(). The example below shows this.

```
Iterator<char*> dayIter(&theWeekDays);
for (char* dayName= dayIter.first(); dayIter.moreElems(); dayName=
dayIter.next()) {
   printf("%s\n", dayName);
}
```

3.3.6 Array

Arrays are implemented 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<Ply*> *plyArr = new(ma)Array<Ply>(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 index.

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.

Examples:

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

```
typedef double context_dependent_measure;
parametric_volume_3d_element_coordinate_system *pv3d;
// The entity euler_angles have an array that is retrieved
// by the method ea->get_angles();
euler_angles *ea = pv3d->get_eu_angles();
Array<context_dependent_measure> *angles = ea->get_angles();
context_dependent_measure angle1 = angles->get(1);
context_dependent_measure angle2 = angles->get(2);
context_dependent_measure angle3 = angles->get(3);
```

3.3.7 Multiple inheritance, sub- and supertypes

Working with objects of classes that have sub and supertypes in C++ may be challenging. In the following we will address these problems.

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 have several subtypes, it will be pointers to objects of these subtypes that are returned. The following example shows such a situation where the program shall handle the different subtypes differently. The solution is to implement first() and next() methods that also returns the object type of the object returned.

```
representation_context *context;
Set<representation*> *reps;
reps = context->get_representations_in_context();
// entityType is a generated enumeration with one value for each
// entity/object class in the schema.
entityType repType;
Iterator<representation*, entityType> repIter(reps);
for (void* rep = repIter.first(&repType); rep; rep =
repIter.next(&repType)) {
   // entityType have value et_<enity name>
   if (repType == et_node) {
      // rep is a node * and the cast below is safe.
      node *n = (node*)rep;
      char *desc = n->get_description();
      char *id = n->get_id();
      char *name = n->get_name();
   } else if (repType == et_fea_model_3d) {
      ...
   } else if (repType == et_point_representation) {
      ...
   }
}
```

In other cases one does not need to handle each subtype case individually like the example over. For example if the task is to invoke a method where the supertype is required, convert from sub to supertype. This is a problem especially when object classes with multiple inheritances are involved. One simple solution can be as follows:

```
for (void* rep = repIter.first(&repType); rep; rep =
repIter.next(&repType)) {
   Representation *myRep;
   if (repType == et_node) {
      node *n = (node*)rep; myRep = n;
   } else if (repType == et_fea_model_3d) {
      fea_model_3d *fm = (fea_model_3d*)rep; myRep = fm;
   } else if (repType == et_point_representation) {
      ...
   }
   executeTheMethod(myRep);
}
```

This is a cumbersome solution especially when there are many subtypes. Therefore the C++ generator has generated a method called supertype_cast that perform safe casting from sub- to supertype. The example below shows how supertype_cast is used:

```
for (void* rep = repIter.first(&repType); rep; rep =
repIter.next(&repType)) {
   representation *myRep;
   myRep = (representation*)supertype_cast(et_representation,
            rep, repType);
   executeTheMethod(myRep);
}
```

The problem with sub- and supertypes arises also when executing get methods. The C++ generator generates two different versions of the get methods when subtypes are possible. The example below shows the declaration of time_interval_assignment. time_interval has subtypes and threfore you have two alternatives for get_assigned_time_interval:

1. `time_interval*get_assigned_time_interval()` – the supertype_cast is here executed behind the scene and one is sure that it is a time_interval pointer that is returned.
2. `void *get_assigned_time_interval(entityType *etp)` – use this version when you want to handle the different subtypes individually.

```
class time_interval_assignment :  public dbInstance
{
public:
   time_interval*    get_assigned_time_interval();
   void *            get_assigned_time_interval(entityType *etp);
};
```

3.3.8 Finding references to objects

The Express function USEDIN returns each object that uses the specified object in a specified role. The C++ Express API has a similar construction. When object A is referencing another object, object B, by putting the appropriate attribute of object A, a pointer from B back to A is set. By this it is easy to find all objects that are referencing object B. In other words: Every object has a linked list of all other objects referencing itself. The API has a special iterator to traverse this linked list, the ReferencesIterator.

```
node *n;
ReferencesIterator<nodal_freedom_action_definition*, entityType>
nfaIter(n, et_nodal_freedom_action_definition);
nodal_freedom_action_definition *nfad
for (nfad = nfaIter.first(); nfad; nfad = nfaIter.next()) {
   ....
}
```

If the program only need one and/or there only exist one object of the specified type the getFirstReferencing(<wanted type>, bool <subtypes>) can be used. Example:

```
// representation have identification stored as the
// attribute_value of a id_attribute object pointing to it
// by the attribute identified_item.
void representation::put_identification(char *identification)
{
   id_attribute *ia = getFirstReferencing(et_id_attribute);
   if (ia == NULL) {
      ia = newObject(id_attribute); ia->put_identified_item(this);
   }
   ia->put_attribute_value(identification);
}
```

If the problem to be addressed requires that you treat the different subtypes individually or you should handle several object types, you must use related AllReferencesIterator. AllReferencesIterator has two constructors, the first one with no parameters that will return all objects when traversing. The second constructor gets an array of object types where the last value is et_indeterminate, this specifies which object types are to be returned from first()/next() methods. Example:

```
node *n;
entityType ct;
AllReferencesIterator<entityType> nodeRefs(n);
for (void *obj = nodeRefs.first(&ct); obj; obj = nodeRefs.next(&ct))
{
   if (ct == et_nodal_freedom_values) {
      nodal_freedom_values *nfv = (nodal_freedom_values*)obj;
   } else if (ct == et_surface_3d_element_representation) {
      surface_3d_element_representation *s3der =
     (surface_3d_element_representation*)obj;
   } else if (ct == et_nodal_freedom_action_definition) {
      nodal_freedom_action_definition *nfad =
```

```
        (nodal_freedom_action_definition*)obj;
    } else if (ct == et_node_set) {
        node_set *nodeSet = (node_set*)obj;
        int dimension = nodeSet->get_dim();
        char *nodeSetName = nodeSet->get_name();
    } else {
        ....
    }
}
```

3.4 Store and retrieve objects in the database

The first use case for the C++ Express API is translators of native CAD/CAE data to and from ISO-10303 AP209. In this use case data are read from native CAD/CAE files into the memory and translated to AP209 model and then written to an EDM model in one operation. With respect to database access this is a simple task. Therefore two methods for database access are implemented and tested.

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

When all objects of a database model are read into the memory, the following procedure can be used for accessing the objects: By executing the method defineObjectSet before copying the objects into the memory, the library creates a set of objects of the class specified in defineObjectSet. After reading all objects into memory one can create an iterator and link it to the set by the method getObjectSet.

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

List<product*> *productsList =
(List<product*>*)myModel.getObjectSet(et_product);
entityType productType, ct;

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

3.5 Express types and constructs not implemented

The following EXPRESS constructs are either not or imperfect implemented in the AP209 API:

- 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 attribute is correct.

- Derived attributes – the implementation of derived attributes have a similar weakness as inverse attributes have. They are only correct 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, the derived attribute is not changed.
- Functions – not implemented
- Rule validation – not implemented.

3.6 Save objects in the database and reuse of memory

The C++ Express API creates objects in memory and in some applications number of objects can be so many that lack of memory becomes a problem. To cope with this the concept of Sub Model is introduced. A Sub Model is always created with a link to a Main Model and objects created in both are stored in the same database model when written to the database. The idea behind the Sub Model is that it has its own memory area that can be released and reused when the objects in the Sub Model are written to the database. The example below shows the use of class SubModel:

```
CMemoryAllocator ma(0x100000), sma(0x100000);

AP209_MIM_model mainModel(rep, &ma);
SubModel subModel(&mainModel, &sma);
mainModel.open("ourModel", sdaiRW);

// Create fea_model_3d feam in the main model. It will be updated
// during the entire execution of the program.
fea_model_3d *feam = newObjectInMain(fea_model_3d); // created in mainModel

for (int i=0; i < 5; i++) {
   node_set *theNodeSet = newObjectInSub(node_set); // created in subModel
   for (int j=0; j < 4; j++) {
      node *n = newObjectInSub(node); // created in subModel
      char name[256];
      sprintf(name, "node_%d_%d", i, j);
      n->put_name(sma.allocString(name));
      theNodeSet->put_nodes_element(n);
   }
   subModel.writeAllObjectsToDatabase();
   MIM::entityType et;
   // Create a reference to theNodeSet object in main model memory
   node_set *nodeSetCopyInMain = (node_set*)mainModel.clone(theNodeSet);
   feam->put_items_element(nodeSetCopyInMain);
   // Release memory in subModel by reset().
   subModel.reset();
   // Now subModel memory can be reused
}
// Write feam to the database containing refereces to the 5 node_set
// created in the loop above.
mainModel.writeAllObjectsToDatabase();
mainModel.close();
```

4 Implement Business Object API using C++ Express API

4.1 Introduction

In the STEP context a Business Object Model is mapped to an Application Interpreted Model (AIM). Its purpose is to hide some of the complexity of the standardized AIM models.

A Business Object API is an API where the programmer writes his program using business objects and if it creates business objects, it is the corresponding AIM objects that at the end are stored in the database. Similarly if one implements a program that reads business objects from a database, the objects in the database are AIM objects that are translated to business objects when the API reads objects from the database.

When implementing a Business Object API there are several options: One option is first to create business objects and when these are committed to the database, translate them to the corresponding AIM objects in a batch operation before storing them in the database. Another option is to create the corresponding AIM objects "on the fly" when a business object is created. We have chosen the last option. The C++ Express API generator has several functions that support implementation of Business Object API using "on the fly" AIM object instantiation.

Central for these functions are the definition of the mapping between Business Object Model (BOM) and Application Interpreted Model (AIM). This information is stored in an Express model that the C++ generator reads when generating the different C++ files.

Another central concept in the implementation and use of the Business Object API is the Business Object Factory. This is an object that has methods for creating the different business objects.

In the following sections we will describe the mapping between BOM and AIM model, the Business Object Factory and the different generator products that support the implementation.

4.2 Define Business Object Model Mapping

The mapping between BOM and AIM model is specified using the Express model CPP_MAPPING shown in Figure 2 below.

The CPP_MAPPING model has the following entities (object classes):

- Mapping_schema – defines the source and target schemas for the mapping. It also has a list of the names of the complex entities that shall be generated.
- Entity_def – defines the entities (object classes) of the schemas.
- Entity_mapping – defines one to one, one to many and many to one mapping between entities.
- Reference_manual and Manual_section – for specification of reference documentation that is copied the C++ header files.

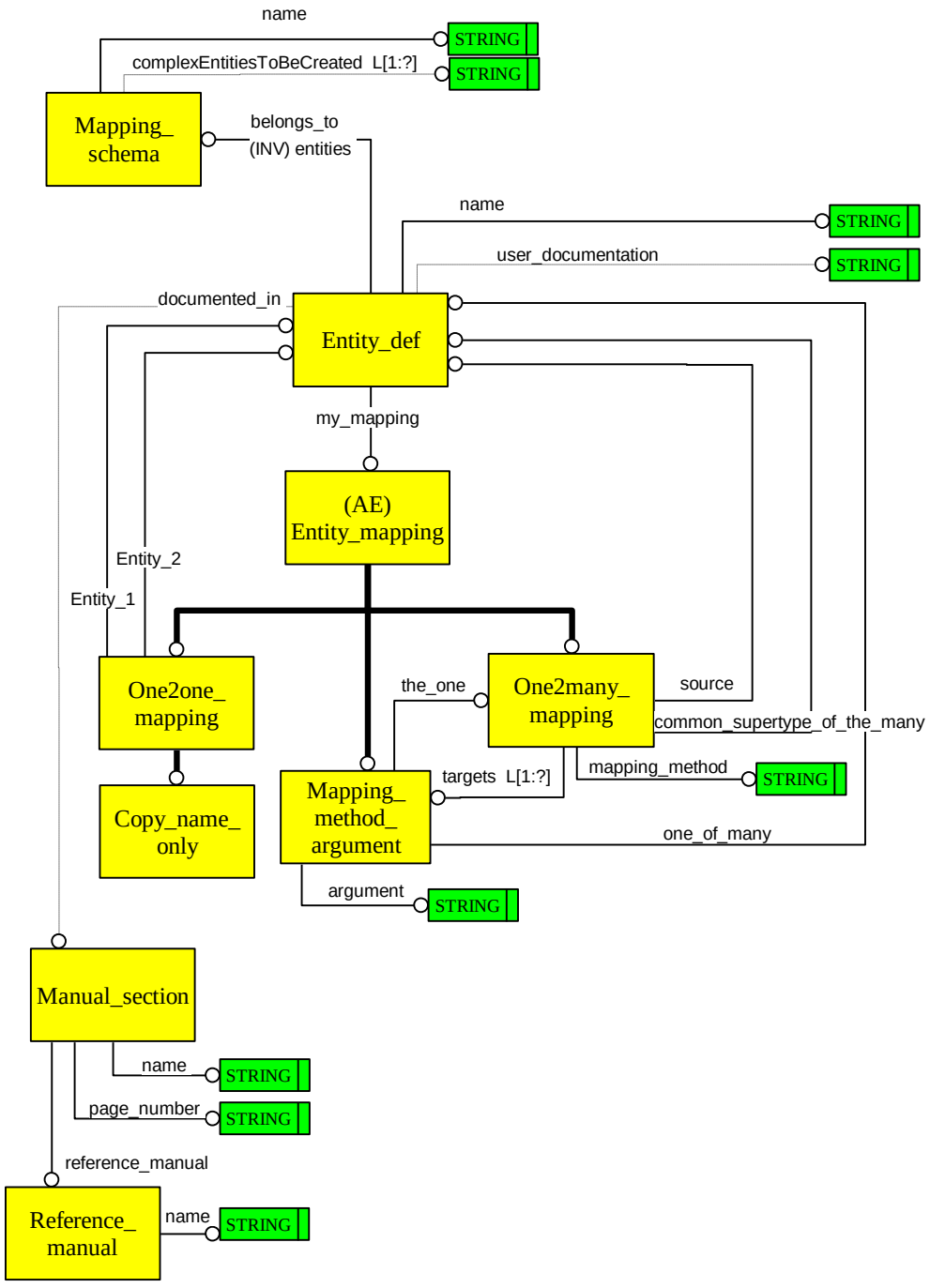


Figure 2 - CPP Mapping

4.3 Implementing Business Object Factory

The Business Object Factory is an object that has methods for creating business objects. Its implementation is supported by the Express C++ generator. The different files generated and their uses are described in the following with the

AP209objectFactory as an example. The first three files are the files that are generated when a standard C++ interface to an Express schema is generated.

- *ap209_bom.hpp* – the C++ definition of the AP209 Business Object Model. The AP209 Business Object Model contains definitions from the AP209e2 AIM model. The Business Object Factory will use both the BOM and the AIM model. To avoid confusion, the AIM objects referenced in the BOM model are not generated.
- *ap209_bom.cpp* – Contains tables defining the attributes of the entities, methods for finding supertype of objects and instantiation of schema objects.
- *AP209_bom_entityTypes.h* – implements the
- *ap209_bom_mapping_tables.cpp* – C++ version of the CPP_MAPPING used to implement BOM to/from AIM mapping.

5 AP209e2 Business Object API

5.1 Mapping between business objects and AIM objects

Business objects are mapped to AIM objects in different ways:

1. One to one – A business objects is always to an AIM object of one specific object class. Example: FeaMaterial always maps to element_material and vice versa.
2. One to many – A business object can map to several AIM object classes. Example: SurfaceElement may map to either axisymmetric_surface_2d_element_representation, plane_surface_2d_element_representation or surface_3d_element_representation. The three AIM object classes will always map to business object class SurfaceElement.
3. Many to one – Many business objects can map to one AIM object class. In this case the AIM object will have attributes that either is or links to the identifier for the business object class.

In the AP209e2 API the actual mapping from AIM to BOM is executed when the user program retrieves attributes from the BOM objects.

5.2 Retrieving Business Objects

When retrieving a business object from the database the client program must declare database and repository objects (see section 3.2 for description). The AP209e2 API has a special object class for AP209 models, AP209_MIM_Model. In addition to the model described in section 3.2, it has reference to the AP209 schema and mapping tables describing the mapping from business objects to/from AIM objects. In the following example, we will discuss a simple program that opens a AP209 model, reads the objects into memory and creates an iterator to iterate over all Analysis business objects.

```
// declare a AP209_MIM_Model with reference to the repository rep
CMemoryAllocator ma(102400);
AP209_MIM_model myModel(rep, &ma);

// open the model in read only mode
myModel.open(modelName, sdaiRO);

// Specify that it shall be possible to iterate over the Analysis
// business objects after the AIM objects are read from the database.
// defineBOMobjectSet use the BOM/AIM mapping tables of myModel.
myModel.defineBOMobjectSet(et_Analysis);

// read all AIM objects into memory.
myModel.readAllObjectsToMemory();

// declare the Anylsis iterator.
Iterator<BO_Analysis*, BOM::entityType>
aIter((Set<Analysis*>*)MIM_model.getBOMobjectSet(et_Analysis));
```

```
// iterate over all Anlysis business objects.
for (Analysis *a = aIter.first(); a; a = aIter.next()) {
   // handle each Anlysis object.
}
```

Iterating over business objects that refer to one specified business object is made possible by the business object reference iterator BOM_ReferencesIterator. To demonstrate the use of BOM_ReferencesIterator, we continue with the example above. An Analysis object is referenced by all its AnalysisVersion objects by the AnalysisVersion.of_analysis attribute.

```
// iterate over all Anlysis business objects.
for (Analysis *a = aIter.first(); a; a = aIter.next()) {
   // handle each Anlysis object.
   // iterate over all analysis versions of a.
   BOM_ReferencesIterator<BO_Analysis_version*, BOM::entityType>
   refIter(a->getMIM(), et_AnalysisVersion);

   // iterate over all analysis versions of a.
   AnalysisVersion *av;
   for (av = refIter.first(); av; av = refIter.next()) {
      char *desc = av->get_description();
      char *id = av->get_version_id();
      // handle each analysis version object.
   }
}
```

6 AP209e2 API Getting Started Tutorial

The AP209e2 API tutorial is organized in such a way that it takes the reader through an example. The main intention is support development of translators that convert from native representation to AP209e2 populations. Hence the focus is on constructing business objects and saving them to database. Reading objects from the database for further processing is not exemplified.

6.1 Initialization

There is, as always, an initialization stage before any instances can be created. Let us assume that a database already exists, and that a data model has already been created. The first action after opening the database is to construct a memory allocator, since such an allocator is required in many subsequent operations. Then the data model is opened before the object factory is constructed. Please note that the AP209_BOM_model class is deprecated. It is a remnant from the initial version of the API, and will be absent in the next version of the API.

```
Database db("db_directory", "db_name", "password");
db.open();
Repository rep(&db, "rep_name");

CMemoryAllocator ma(1024*1024);

AP209_BOM_model bomModel(&rep, &ma); // deprecated
AP209_AIM_model mimModel(&rep, &ma);
mimModel.open("model_name", sdaiRW);

AP209objectFactory objFact(&bomModel, &mimModel);
```

6.2 The object factory

All business objects have a corresponding factory method (new_<Business object name>. The signatures of the factory methods are found in the *AP209objectFactory.h* header file. There is one parameter for each mandatory attribute in the corresponding entity of the underlying BOM EXPRESS schema.

```
BO_View_context *vcdesign =
      objFact.new_BO_View_context("fea_analysis_domain","design_stage
");
```

Some pure A209e2 AIM classes are exposed by the business object model. These classes do not , as of this version of the API, follow the same pattern for construction as genuine business objects. The `newAIMobject` macro is applicable for such classes. The macro depends on a Model instance named *mim*, which has to be obtained from the factory beforehand. The attributes of the AIM objects must be populated one by one by means of their corresponding *put* methods.

```
Model *mim = objFact.getmim();
fea_area_density *fad =  newAIMobject(fea_area_density);
fad->put_name("area density");
scalar ad = 7.88;
```

```
   fad->put_fea_constant(ad);
```

6.3 Aggregate constructors

Templates for the four different aggregate types are offered by the API. The memory allocator has to be handed over to the aggregate constructors, so that aggregates can be placed in the very same cache buffer as the instances. Please noet that the last parameter of the aggregate constructor is neither the expected size nor the maximum size of the aggregate. Memory for aggregates is allocated in chunks and the last parameter represents the number of aggregate elements in each chunk.

long eioc = 1;

```
Set<analysis_category_name_enum> *categories = new(&ma)
Set<analysis_category_name_enum>(ma, sdaiENUMERATION, eioc);
categories-
>add(analysis_category_name_enum_LINEAR_STATIC_ANALYSIS,&ma);
BO_Analysis *analysis = objFact-
>new_BO_Analysis("analyis_id","analysis_name",categories);
```

Please note that when the elements of an aggregate are instances of BOM objects rather than AIM objects the method *BO_add* has to be applied instead of the usual *add* method.

```
   BO_View_context *context =
objFact.new_BO_View_context("my_doamin","my_life_cycle_stage");
   Set<BO_View_context*> *context_set = new(&ma)
Set<BO_View_context*>(&ma, sdaiINSTANCE, 1);
   context_set ->BO_add(context,&ma);
```

6.4 Constructor sequence

Although it is feasible to "repair" incompletely populated instances by means of *put* methods after an instance has been created, it is a recommended practice to construct an instance completely in the first place. This means that the sequence of object factory calls has to be done in such an order that instances which are required as parameters of a specific constructor have to be constructed beforehand. In the example below the `BO_Analysis_version_view` instance is dependant of the `BO_View_context` instance and the `BO_Analysis_version` instance. The `BO_Analysis_version` instance is in its turn dependant of the `BO_Analysis` instance

long eioc = 1;

```
Set<analysis_category_name_enum> *categories = new(&ma)
Set<analysis_category_name_enum>(ma, sdaiENUMERATION, eioc);
categories-
>add(analysis_category_name_enum_LINEAR_STATIC_ANALYSIS,&ma);
BO_Analysis *analysis =
objFact.new_BO_Analysis("analyis_id","analysis_name",categories);
BO_Analysis_version *analysis _version =
objFact.new_BO_Analysis_version("analysis_version_id", analysis);
BO_View_context *context =
objFact.new_BO_View_context("my_doamin","my_life_cycle_stage");
BO_Analysis_version_view *avw =
```

```
objFact.new_BO_Analysis_version_view("analysis_version_view_id",conte
xt, analysis _version);
```

6.5 Redundancy

Although there is a built in mechanism in the API to avoid duplicate instances in the population, it is applied only for a few business object types. The general rule is that the application (translator) is responsible for the reuse of instances in order to avoid redundancy in the population. Contexts, categories, and units are handled by the built in mechanism. So the two subsequent and identical calls to the factory will result in only one AIM instance of type `product_definition_context` in the database.

```
BO_View_context *contex1t =
objFact.new_BO_View_context("my_doamin","my_life_cycle_stage");
BO_View_context *context2 =
objFact.new_BO_View_context("my_doamin","my_life_cycle_stage");
```

Likewise two subsequent calls two the factory result in only one single `mass_unit+si_unit` instance in the database.

```
BO_Predefined_SI_unit *mu1 =
objFact.new_BO_Predefined_SI_unit(si_unit_enum_KILOGRAM);
BO_Predefined_SI_unit *mu2 =
objFact.new_BO_Predefined_SI_unit(si_unit_enum_KILOGRAM);
```

Two identical calls to the `BO_Analysis_version_view` factory method will, however, result in two identical instances, both with the id `avv1", in the database.`

```
BO_Analysis_version_view *avv1 =
objFact.new_BO_Analysis_version_view("avv1",context, analysis
_version);
BO_Analysis_version_view *avv2 =
objFact.new_BO_Analysis_version_view("avv1",context, analysis
_version);
```

6.6 Persistence, validation and export

Once all objects have been constructed, they are flushed to the persistent database in one shot.

```
mimModel.writeAllObjectsToDatabase();
```

The validation report for the stored data model is obtained in the following way:

```
EDMLONG rstat = 0;
SdaiRepository repId = 0;
SdaiInstance valErrId = 0;
EDMLONG validateOptions = FULL_VALIDATION | FULL_OUTPUT;
rstat = edmiGetRepository("rep_name",& repId);
rstat = edmiValidateModelBN(repId,"model_name", "valresult.txt",
validateOptions, NULL, NULL, &valErrId);
```

The export to P21 and P26 files is done in the following way:

```
EDMLONG ne,ne;
```

```
SdaiErrorCode sdaiError;
rstat = edmiWriteStepFile("rep_name", NULL, "model_name", "p21.stp",
NULL, NULL, 0, 8, &nw, &ne, &sdaiError);
rstat = edmiWriteHDF5File(("rep_name",
"model_name","p26.h5","p26_dia.txt",0);
```

Finally close the database.

```
rstat = edmiCloseDatabase("");
```

6.7 The coherent example

This small example opens a database, creates a new data model (after deleting a possible existing one). Then the object factory is used to create and instance of type `BO_Analysis_version_view`. All the instances required by the `BO_Analysis_version_view` (directly or indirectly) constructor have been created before the `BO_Analysis_version_view` before the constructor is invoked. The instances are then flushed from the memory cache to the persistent data model. The data model is validated and exported to P21 and P26 files before the database is closed.

```
#include "AP209_API.h"
void Tutorial(void)
{
   try{
      EDMLONG rstat = 0;
//
//    Open database
//
      Database db("database_directory", "db_name", "password");
      db.open();
      Repository rep(&db, "DataRepository");
//
//    Create new data model
//
      SdaiRepository repositoryId;
      rstat = edmiGetRepository("DataRepository",&repositoryId);
      repositoryId = sdaiOpenRepository(repositoryId);
      (void)edmiDeleteModelBN(repositoryId,"model_name");
      (void) sdaiErrorQuery();
      SdaiModel modelId =
      sdaiCreateModelBN(repositoryId,"model_name","AP209_MULTIDISCIPL
      INARY_ANALYSIS_AND_DESIGN_AIM_LF");
//
//    Construct a memory allocator
//
      CMemoryAllocator ma(1024*1024);
//
//    Open the data model
//

      AP209_BOM_model bomModel(&rep, &ma); // deprecated
      AP209_AIM_model mimModel(&rep, &ma);
      mimModel.open("model_name", sdaiRW);
//
//    Construct the object factory
//
```

```
      AP209objectFactory objFact(&bomModel, &mimModel);
//
//    Create an BO_Analysis instance. The set of categories is
created before hand , just in one member.
//
      long eioc = 1;
      Set<analysis_category_name_enum> *categories =
                                          new(&ma)
Set<analysis_category_name_enum>(&ma, sdaiENUMERATION, eioc);
      categories-
>add(analysis_category_name_enum_LINEAR_STATIC_ANALYSIS,&ma);
      BO_Analysis *analysis =
objFact.new_BO_Analysis("analyis_id","analysis_name",categories);
//
//    Create an BO_Analysis_version instance.
//
      BO_Analysis_version *analysis_version =
objFact.new_BO_Analysis_version("analysis_version_id", analysis);
//
//    Create an BO_Analysis_version_view instance. The context is
cretaed before hand.
//
      BO_View_context *context =
objFact.new_BO_View_context("my_doamin","my_life_cycle_stage");
      BO_Analysis_version_view *avw =
      objFact.new_BO_Analysis_version_view("analysis_version_view_id"
      ,context,  analysis_version);
//
//    Flush all instances in cache to the database
//
      mimModel.writeAllObjectsToDatabase();
//
//    Validate the data model
//
      SdaiRepository repId = 0;
      SdaiInstance valErrId = 0;
      EDMLONG validateOptions = FULL_VALIDATION | FULL_OUTPUT;
      rstat = edmiGetRepository("DataRepository",& repId);
      rstat = edmiValidateModelBN(repId,"model_name",
"valresult.txt", validateOptions, NULL, NULL, &valErrId);
//
//    Export to P21 and P26 files.
//
      EDMLONG nw,ne;
      SdaiErrorCode sdaiError;
      rstat = edmiWriteStepFile("DataRepository", NULL, "model_name",
      "p21.stp", NULL, NULL, 0, 8, &nw, &ne,     &sdaiError);
      rstat = edmiWriteHDF5File("DataRepository",
"model_name","p26.h5","p26_dia.txt",0);
//
//    Close database
//
      rstat = edmiCloseDatabase("");

   } catch(CedmError *e) {
      long rstat = e->rstat;
      if (e->message) {
        char *errTxt = e->message;
      } else {
```

```
        char *errTxt = edmiGetErrorText(rstat);
      }
      delete e;
    } catch(int thrownRstat) {
      long rstat = thrownRstat;
    }
}
int main(int argc, char* argv[])
{
  Tutorial();
}
```

6.8 Second example

In this example we will present a small function that creates a product structure (PS) in an analysis context. The product structure is for a table that has one top and four legs. Each leg has a rod and a cap. To each node in the PS the create time will be assigned.

The create PS function is implemented as the method createTheExample() of the class PS_Builder. PS_Builder inherits AP209objectFactory. By that all methods for creating and updating AP209 Business Objects becomes available for PS_Builder methods.

To create a node in a Product Structure (PS) the AP209 Business Object Model (BOM) use BO_Part, BO_Part_version and BO_Part_version_view. See Figure 3 and Figure 4. To create the PS the nodes are linked together with BO_Next_higher_assembly. See Figure 5.

createTheExample() starts with creating default organization and application context. The next step is to create the PS. BO_Part together with BO_Part_version is created with createPartWithVersion(). Assemblies are created with createAssembly () and leaf nodes are created by createPart() and the link between nodes are simply created by new_BO_Next_higher_assembly().

In the listing below one can see the code for creating BO_Next_higher_assembly:

```
new_BO_Next_higher_assembly(table, top->toSelect(), NULL);
```

The toSelect() method is used because the second parameter of new_BO_Next_higher_assembly is BOM::part_instance_or_version_view_select*.

The last step of createTheExample() is to create a BO_Date_time called BO_now and assign it to the nodes. The graphical view of BO_Date_time is shown in Figure 6.

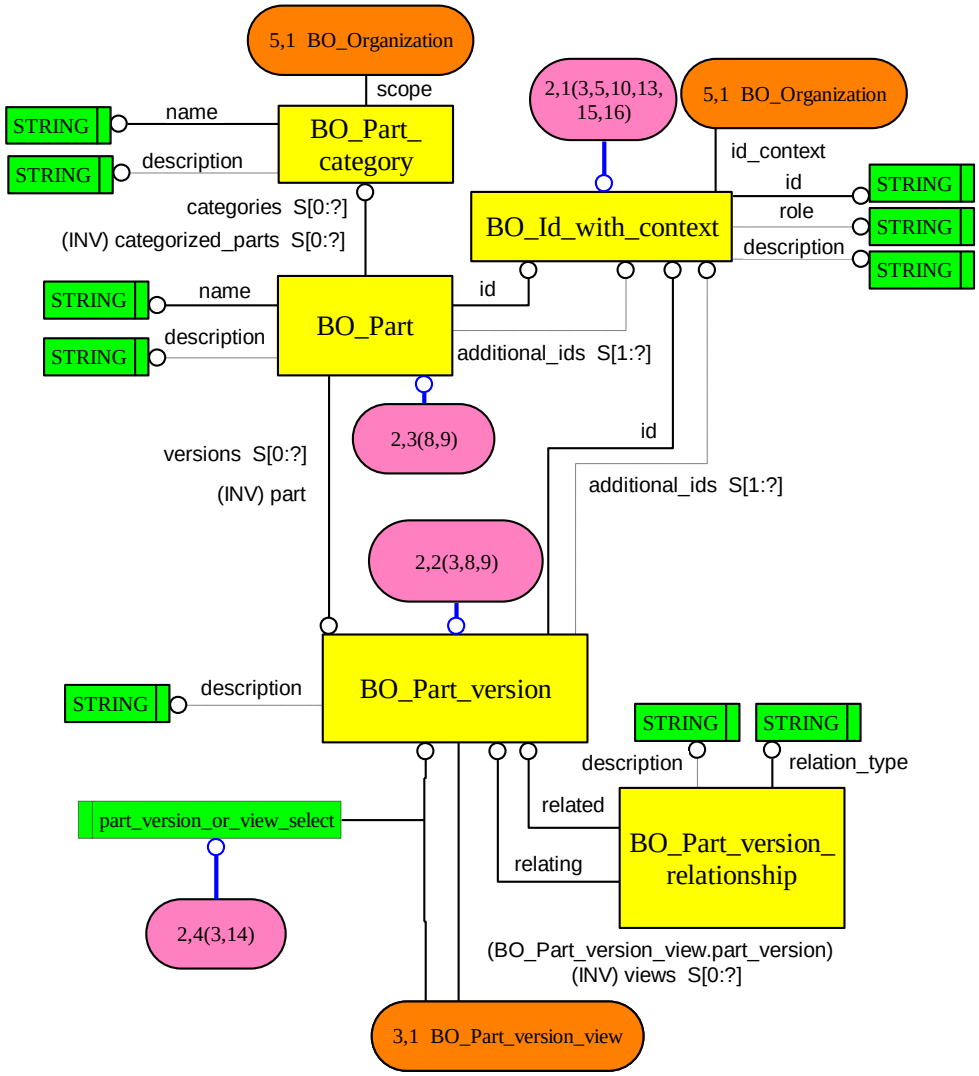


Figure 3 – BO_Part/BO_Part_version/BO_Part_version_view relationship.

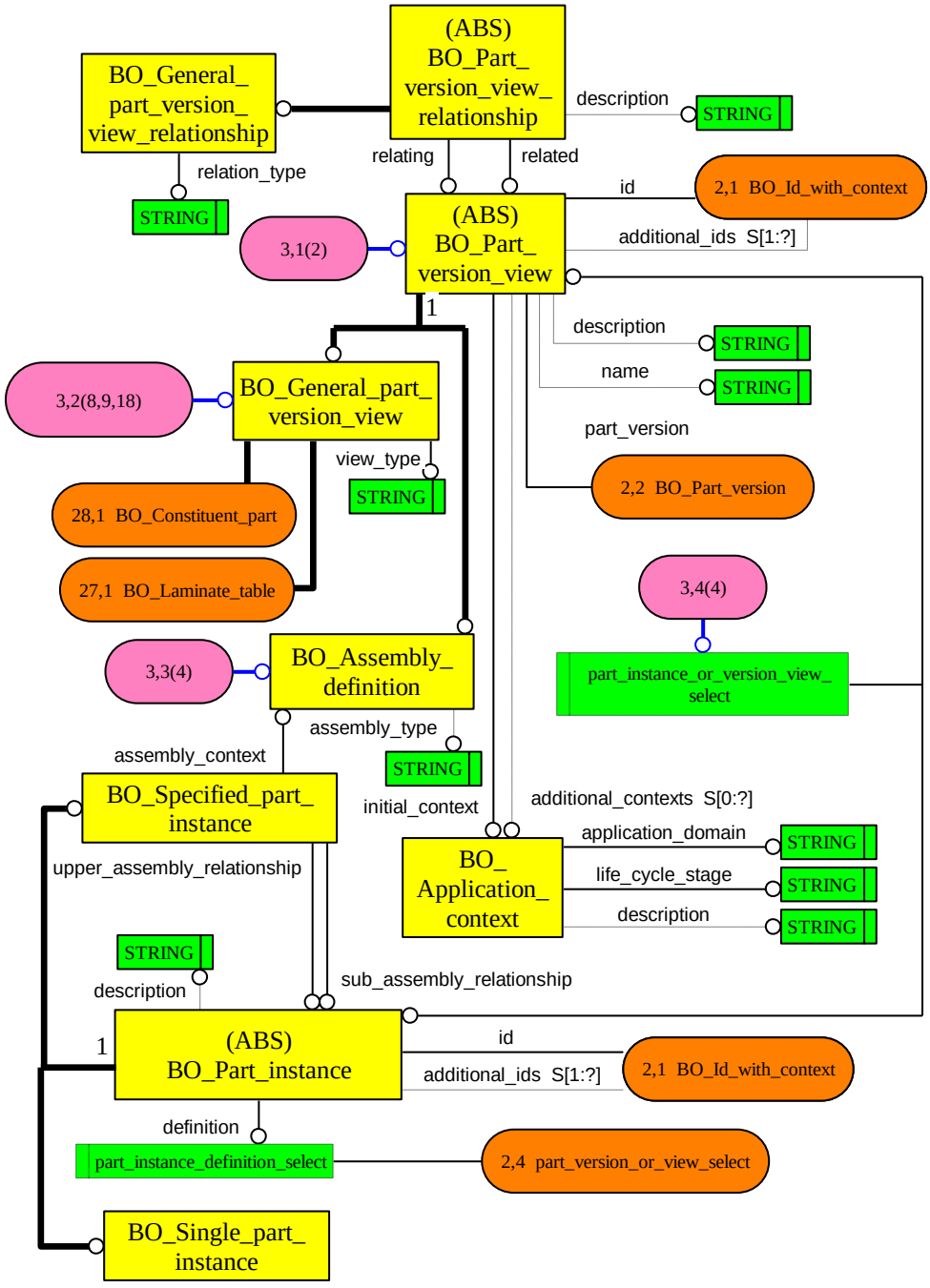


Figure 4 – BO_Part_version_view

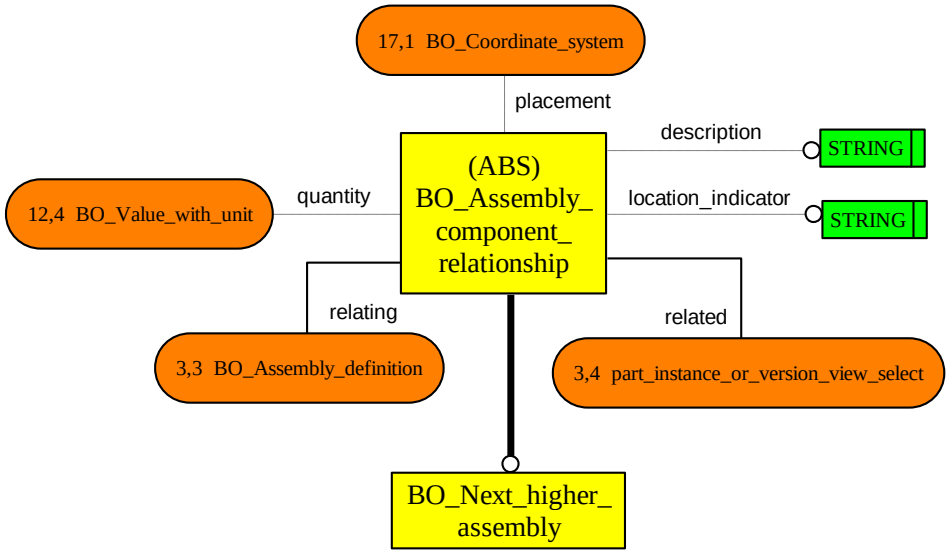


Figure 5 – Assembly structure

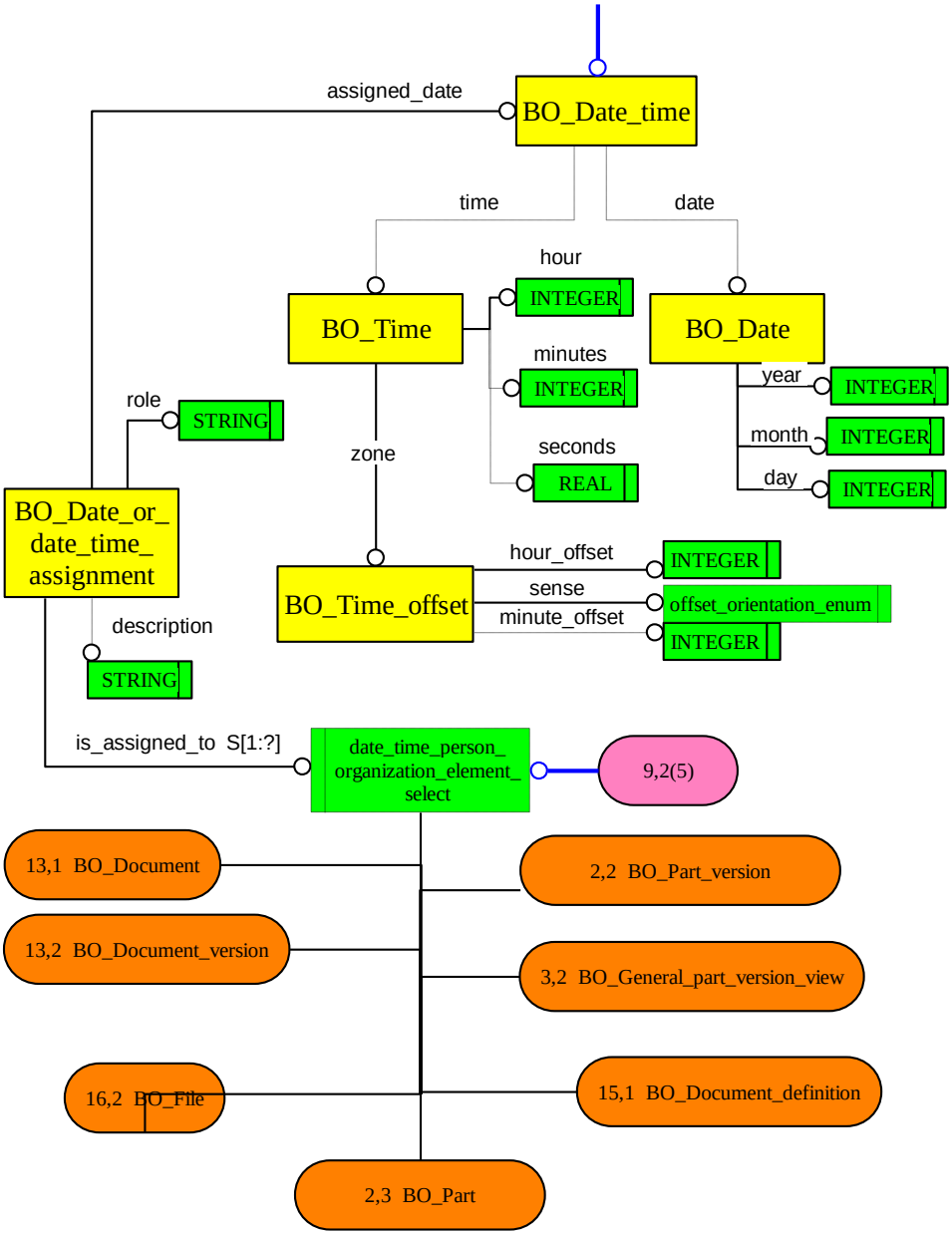


Figure 6 – BO_Date_time

6.8.1 Second example – listing

```
// Example_2.h

/*=============================================================================================*/
class PS_Builder : public AP209objectFactory
/*=============================================================================================*/
{
   BO_Organization                 *ourOrganization; // the default organization
   BO_Application_context          *ourContext;      // the default application context
   BO_Time_offset                  *ourTimeOffset;

   // createPartWithVersion is used by createAssembly and createPart when building the PS.
   BO_Part_version                 *createPartWithVersion(char *id, char *name);
public:

                                   PS_Builder(AP209_MIM_model *mim) ;

   void                            setOrganization(BO_Organization *org) { ourOrganization = org; }
   void                            setApplicationContext(BO_Application_context *ac) { ourContext = ac; }
   void                            setTimeOffset(BO_Time_offset *to) { ourTimeOffset = to; }
   // createAssembly creates PS node that has child nodes
   BO_Assembly_definition          *createAssembly(char *id, char *name);
   // createPart creates PS leadf node that does not have child nodes
   BO_General_part_version_view    *createPart(char *id, char *name, char *viewName);

   // createTheExample() is the main function that builds the PS.
   void                            createTheExample();
};
```

```
// Example_1.cpp

#include "stdafx.h"
#include "Example_2.h"


/*=============================================================================================*/
BO_Part_version *PS_Builder::createPartWithVersion(char *id, char *name)
/*=============================================================================================*/
{
   BO_Id_with_context *theId = new_BO_Id_with_context(id, "Part id", ourOrganization);
   BO_Part *p = new_BO_Part(theId, name, NULL, NULL);
   BO_Id_with_context *theVersionId = new_BO_Id_with_context("1", "Part version id", ourOrganization);
   BO_Part_version *pv = new_BO_Part_version(theVersionId);
   p->put_versions_element(pv);
   return pv;
}

/*=============================================================================================*/
BO_Assembly_definition *PS_Builder::createAssembly(char *id, char *name)
/*=============================================================================================*/
{
   BO_Part_version *pv = createPartWithVersion(id, name);
   BO_Id_with_context *theVersionViewId = new_BO_Id_with_context("1", "Part version view id", ourOrganization);
   return new_BO_Assembly_definition(theVersionViewId, pv, ourContext);
}

/*=============================================================================================*/
BO_General_part_version_view *PS_Builder::createPart(char *id, char *name, char *viewName)
/*=============================================================================================*/
{
   BO_Part_version *pv = createPartWithVersion(id, name);
   BO_Id_with_context *theVersionViewId = new_BO_Id_with_context("1", "Part version view id", ourOrganization);
   return new_BO_General_part_version_view(theVersionViewId, pv, ourContext, viewName);
}
```

```
/*=============================================================================================================*/
void PS_Builder::createTheExample()
/*
   Creates Product Structure (PS) as
   BO_Assembly_definition (node that has child(s)) and BO_General_part_version_view nodes (leaf node)
   and link them together by BO_Next_higher_assembly objects.
   Assign create time to all the nodes in the PS.
=============================================================================================================*/
{
   // When an organization is needed, thePS_Builder will use "ModelAnalysis inc."
   setOrganization(new_BO_Organization("ModelAnalysis inc."));
   // application context is set by the following:
   setApplicationContext(new_BO_Application_context("FEM analysis", "analysis"));
   setTimeOffset(new_BO_Time_offset(1, 0, offset_orientation_enum_OFFSET_BEHIND));

   BO_Assembly_definition *table = createAssembly("i1", "Table");
   BO_General_part_version_view *top = createPart("i2", "Top", "analysis");
   BO_Next_higher_assembly *table_top = new_BO_Next_higher_assembly(table, top->toSelect(), NULL);
   BO_Assembly_definition *leg = createAssembly("i3", "Leg");
   BO_Next_higher_assembly *table_leg = new_BO_Next_higher_assembly(table, leg->toSelect(), NULL);
   BO_General_part_version_view *rod = createPart("i4", "Rod", "analysis");
   BO_General_part_version_view *cap = createPart("i5", "Cap", "analysis");
   BO_Next_higher_assembly *leg_rod = new_BO_Next_higher_assembly(table, rod->toSelect(), NULL);
   BO_Next_higher_assembly *leg_cap = new_BO_Next_higher_assembly(table, cap->toSelect(), NULL);

   struct tm *now;
   time_t ltime;

   time( &ltime ); now = gmtime( &ltime );

   // create BO_Date_time BO_now
   BO_Date_time *BO_now = new_BO_Date_time(new_BO_Date(now->tm_year + 1900, now->tm_mon + 1, now->tm_mday));
   BO_Time *time = new_BO_Time(now->tm_hour, now->tm_min, now->tm_sec, ourTimeOffset);
   BO_now->put_time(time);
   // and link the BO_now to the PS nodes by a BO_Date_or_date_time_assignment
   BO_Date_or_date_time_assignment *dt_asm = new_BO_Date_or_date_time_assignment(BO_now, "create time", NULL);
```

```
   dt_asm->put_is_assigned_to_element(table->toSelect());
   dt_asm->put_is_assigned_to_element(top->toSelect());
   dt_asm->put_is_assigned_to_element(leg->toSelect());
   dt_asm->put_is_assigned_to_element(rod->toSelect());
   dt_asm->put_is_assigned_to_element(cap->toSelect());
}
/*============================================================================================*/
PS_Builder::PS_Builder(AP209_MIM_model *m) : AP209objectFactory(m)
/*============================================================================================*/
{
}

/*============================================================================================*/
void myExceptionHandler(char *message, char *file, int lineNo)
/*
   Using this exption handler makes it easy to break the program when exceptions occur
==============================================================================================*/
{
   throw new CedmError(message, file, lineNo);
}

/*============================================================================================*/
int _tmain(int argc, _TCHAR* argv[])
/*============================================================================================*/
{
   CMemoryAllocator           ma(0x100000);
   long                       rstat = 0;
   EdmiError                  sdaiError;
   SdaiInteger                nbWarnings, nbErrors;

   char* database_dir      = "db";
   char* database_name     = "d";
   char* database_password = "d";
   char* repository_name   = "DataRepository";
   char* model_name        = "testModel";
```

```
    try {
        defineExceptionHandler(&myExceptionHandler);
        Database db(database_dir, database_name, database_password);
        db.open();
        Repository cRepository(&db, repository_name);

        AP209_MIM_model myModel(&cRepository, &ma);
        myModel.open(model_name, sdaiRW);

        // Decalre Product Structure builder
        PS_Builder thePS_Builder(&myModel);

        // Delete model content in case of runnig this program more than once.
        CHECK(edmiDeleteModelContents(myModel.modelId));

        // Build the Product Structure (PS) in memory
        thePS_Builder.createTheExample();

        // Write the PS to the database
        myModel.writeAllObjectsToDatabase();

        // and write the PS to a STEP file.
        CHECK(edmiWriteStepFile(repository_name, NULL, model_name, "Example_1.stp",
        NULL, NULL, 0, 6, &nbWarnings, &nbErrors, &sdaiError));

        printf("Finished OK\n%s.%s written to Example_1.stp", repository_name, model_name);
    } catch(CedmError *e) {
        printf("Error encountered: \"%s\"\n", e->message ? e->message : edmiGetErrorText(e->rstat));
        delete e;
    } catch(int thrownRstat) {
        printf("Error encountered: \"%s\"\n", edmiGetErrorText(thrownRstat));
    }
    int closeRstat = edmiCloseDatabase(repository_name);

    return 0;
}
```

6.8.2 How to build Example_2

Figure 7 – C++ AP209e2 API to EDM shows a sketch of the interface to the EDM database from a C++ program using the AP209e2 API. The sketch shows that the C++ AP209e2 API consists of four parts. These are:

1. AP209 BOM API – AP209 Business Object API. This API maps AP209 Business Objects AP029e2 AIM objects. These objects are read/written by the C++ Express API. The C++ Express API use *EDMinterface*™ for database communication.
2. AP209 AIM API – The AP209 Business Object Model covers only a part of the AP209 functionality. Therefore is the AP209 AIM API necessary for a application program that shall cover all AP209
3. C++ Express API – the schema independent part of the API.
4. *EDMinterface*™ - the complete programmers interface to EDM.

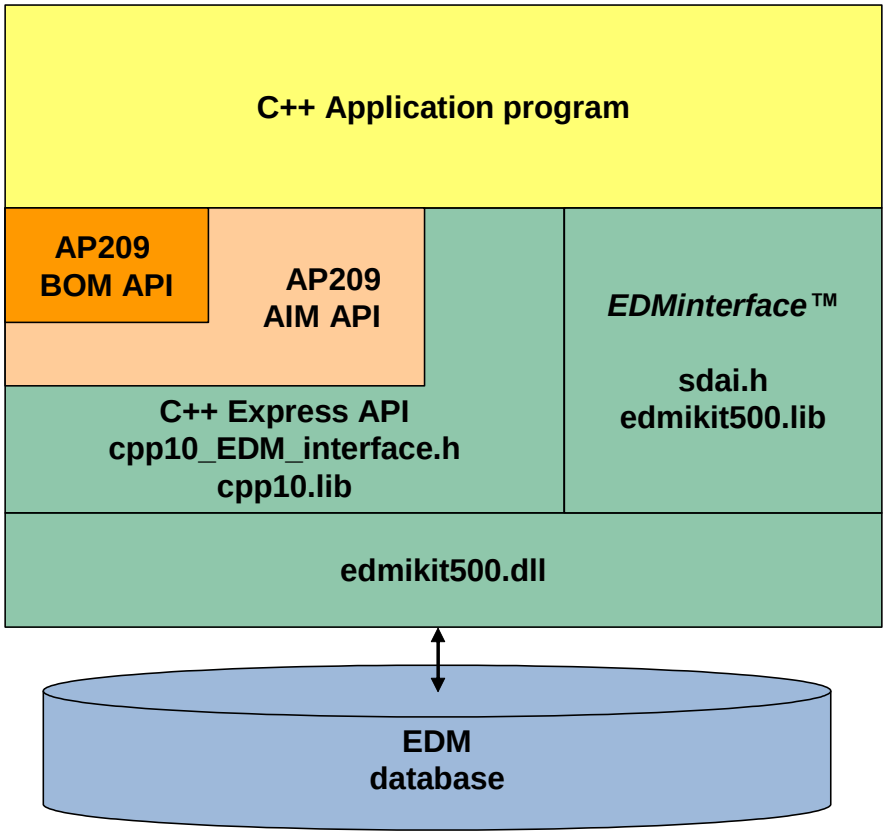


Figure 7 – C++ AP209e2 API to EDM

The AP209e2 API is delivered in three folders:

1. AP209e2_API, the AP209 and C++ Express API specific part of the library.

2. EDMv5.01.0100.02-VS2008, the EDM SDK. The name indicates that it is version 5.01.0100.02 of EDM that is delivered. And the -VS2008 part of the name indicates that this version is generated for Microsoft Visual Studio 2008.
3. Example_2

In the following we will show how a C++ application program defines and links to the different parts of the AP209e2 API. Example_2 use precompiled headers. Therefore are the necessary include statements written in stdafx.h. A closer look at this file:

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

#include "AP209_BOM_entityTypes.h"
#include
"AP209_MULTIDISCIPLINARY_ANALYSIS_AND_DESIGN_MIM_LF_entityTypes.h"
#include "container.h"
#include "AP209_MULTIDISCIPLINARY_ANALYSIS_AND_DESIGN_MIM_LF.hpp"
using namespace MIM;
#include "AP209_BOM.hpp"
using namespace BOM;

#include "AP209dbInterface.h"
#include "AP209objectFactory.h"
```

Explanation of the different include files to stdafx.h:

- sdai.h – definition of *EDMinterface*™
- cpp10_EDM_interface.h – EDM interface for the C++ Express API
- AP209_BOM_entityTypes.h – Runtime type information (RTTI) for the AP209 Business Object Model
- AP209_MULTIDISCIPLINARY_ANALYSIS_AND_DESIGN_MIM_LF_entityTypes.h - Runtime type information (RTTI) for the AP209e2 AIM model.
- container.h – Definitions of the C++ Express API.
- AP209_MULTIDISCIPLINARY_ANALYSIS_AND_DESIGN_MIM_LF.hpp – definition of the AP209 AIM
- AP209_BOM.hpp – Definition of the AP209 BOM API.
- AP209dbInterface.h – Definition of the interface to the AP209e2 AIM EDM database.
- AP209objectFactory.h – Definition of the AP209objectFactory and all the methods for creating AP209 Business Objects

The current version of the AP209 API is only available for Microsoft Visual Studio 2008.

When compiling and linking programs that use the AP209 API one must set the following Visual Studio 2008 Configuration properties:

For compiling: C/C++ -> General -> Additional Include directories:

- ..\AP209e2_API\include
- ..\EDMv5.01.0100.02-VS2008\include

And for linking: Linker -> General -> Additional Library directories:

- ..\EDMv5.01.0100.02-VS2008\win32
- ..\AP209e2_API\lib\win32

Linker -> Input -> Additional Dependencies:

- AP209_BOM_API.lib
- cpp10.lib
- edmikit500.lib

6.8.3 How to run Example_2

Before one can run Example_2 one must create a database, define the AP209e2 AIM Express schema and a data model. That must be done by the *EDMsupervisor*™.

A prerequisite for running *EDMsupervisor*™ is to install an EDM license key. An EDM license key is calculated for the machine where the EDM software shall run. To do this one must run .\EDMv5.01.0100.02-VS2008\win32\InstallLicenseKey.exe and enter the license number and password you get from Jotne. InstallLicenseKey.exe will contact the Jotne license server and have a license key calculated and install it.

After installation of license key one is ready for creating an EDM database by the *EDMsupervisor*™. It is available in .\EDMv5.01.0100.02-VS2008\win32. Before running the Example_2 program you must do the following four steps:

1. Database->Create – Specify
    a. Database folder
    b. Database name
    c. Database password.
2. Schemata->Define Schema - File name: . \AP209e2_API\Express\part409cdts_wg3n2617mim_lf.exp
3. Data->Create->Model – Parameter:
    a. Repository: DataRepository
    b. Schema: ap209_multidisciplinary_analysis_and_design_mim_lf
    c. Model: testModel
4. Database->Close

Last step before running Example_2 is to modify line 108 – 110 of Example_2.cpp with database folder, database name and database password from the Database->Create command.

Then build and run the program!

7 Query the SimDM database

With the SimDM client the user can query AP209 files when they are loaded into the SimDM server as EDM models. Figure 8 below shows the Open SimDM client window for text queries. See [6] - Open SimDM Application - Reference Manual for user documentation. This functionality is also available to programs using a text query web service interface to the SimDM server.

There are three different types of queries. These are:

1. Queries that return information about objects where you specify object id(s).
2. Queries that return information about objects with respect to load cases. You specify both objects id(s) and load case id(s). The load case id(s) are selected from a list of all load case ids in the AP209 model. This list of all the load case ids can by obtained be separate web service or, in the API case, a separate API function.
3. Maximum or minimum tensor value surveys. You specify either maximum or minimum type of survey by an option. The survey finds maximum or minimum by analyzing the specified objects in the specified load cases. One variant of the surveys finds maximum or minimum of all specified objects and load cases. Another variant finds maximum or minimum for each specified object over all specified load cases.

The following queries are available:

Query name	**Type of query**
Model Query	1
Node Query	1
Element Query	1
Curve Element Property Query	1
Surface Element Property Query	1
Point Element Property Query	1
Directionally explicit Element Property Query	1
Material Property Query	1
Constraint Element Query	2
Constraint control analysis Query	2
Applied Load Query	2
Displacement Query	2
Stress and Strain Query	2
Element Forces from Node Force Balance Query	2
Displacement Survey	3
Stress or Strain Survey	3
Element Forces from Node Force Balance Survey	3

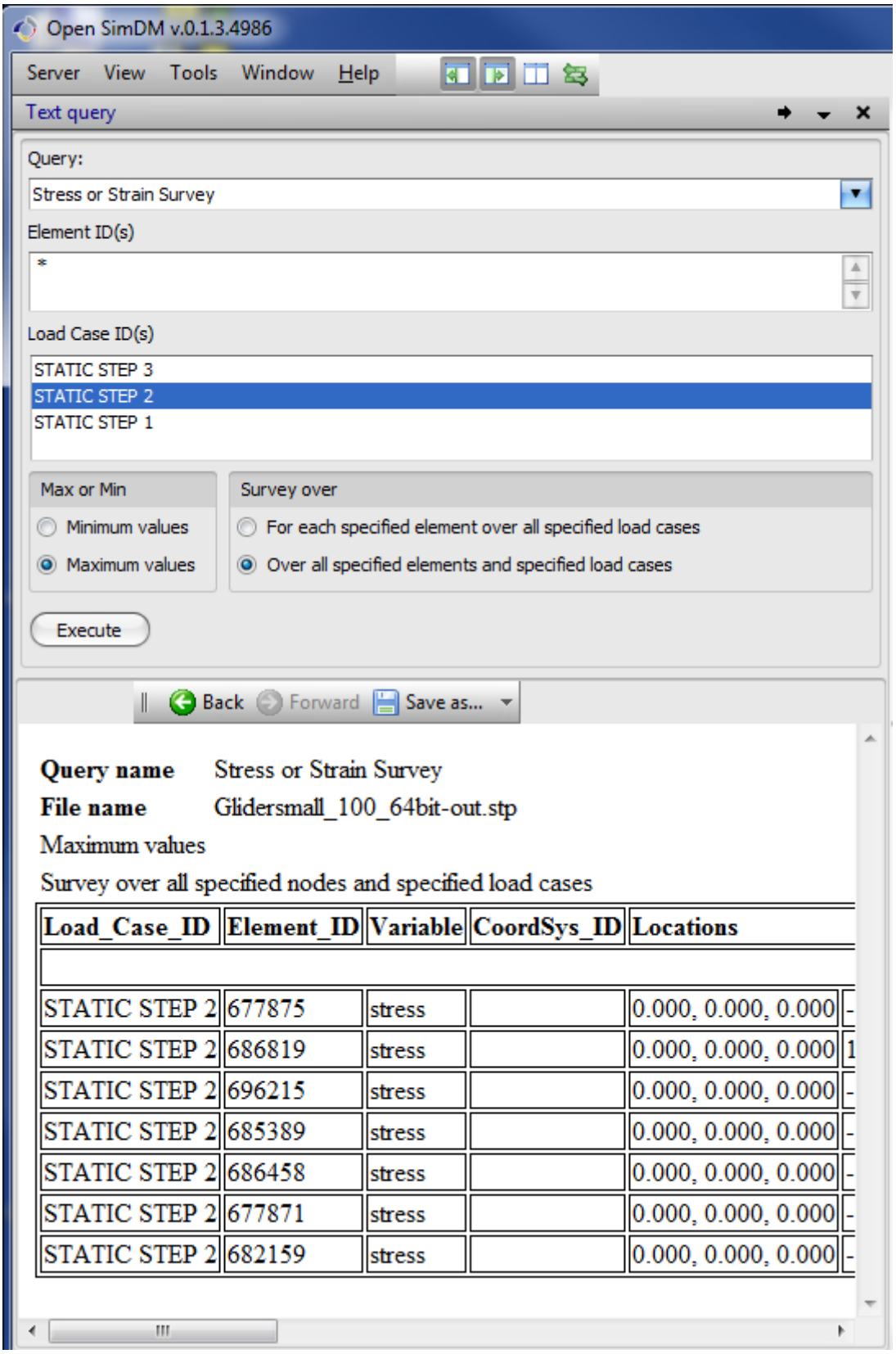


Figure 8 - Open SimDM client text query window

The queries are executed by a generic query web service where the first parameter is a number specifying which query that shall be executed. The name of this generic web service is SimDMobjectViewer and has the following list of parameters:

1. queryIndex – Specifies which query to be executes. Starts by 0 for Model Query, 1 for Node Query, 2 for Element Query etc.
2. objectIds - Object ids. The parameter is a list of strings where the first string are all object ids that are input parameters to the query. The string of object ids shall be the only element in the list. The ids must be separated by space; ids that contain spaces must be enclosed by single quotes. Numerical ids can be specified as a range with a dash between the minimum and the maximum range value. If one single "*" is specified, all objects in the queried database model is searched.
3. loadCases – String containing load case ids. Relevant only for query type 2 and 3.
4. firstRow – For future extension. Always 1 now.
5. maxNoOfRows – For future extension. Always all rows now.
6. options – Only relevant for query type 3.

The return value of the query is a HTML string that can be displayed to the user in a HTML viewer.

Upon request to support@jotne.com you can get a C# project that is an example using this web service.

Appendix A Generated files for AP209e2 API

The implementation of the AP209e2 API is supported by generated C++ header and implementation files for the AP209e2 BOM and AIM schemas. *EDMsupervisor*™ has two commands for generating these files. This Appendix describes these two commands:

1. Schemata->Generate Interface->Cpp 2010
2. Schemata->Generate Interface->Cpp 2010 mapping support

Cpp 2010

Generate Cpp Interface optimized for high speed and large schemata.

Menu path: Schemata>Generate Interface>Cpp 2010

Generates a C++ EXPRESS early binding interface for a specified Express schema. The actual Express schema must exist as a **dictionary model** in the *EDMdatabase*™ before this command can be successfully performed.

This command is initially developed for a project that started in year 2010. Therefore is the name of the command Cpp 2010.

The generated C++ EXPRESS early binding should be compiled by a C++ compiler as part of the actual C++ program using this generated interface.

The command Schemata>DefineSchema can be used to create a **dictionary model** of an Express schema

Arguments:

Schema:	Specify the name of the schema in the *EDMdatabase*™ to generate a C++ early binding interface for. Schema names are case insensitive. When activating the **Select** button, the names of all existing schemata will be displayed in the related selection list.
Cpp interface output directory:	Location for storing of C++ source. When activating the **Select** button, a window is opened to allow browsing the directory structure and select the desired directory.
Namespace:	Unique name for namespace (if omitted - name of schema will be used instead)

Options:

accumulating command output:	A global option that appends the command output to the *EDMsupervisor*™ output window. Otherwise the output window is refreshed and only the last command output is displayed.

Cpp 2010 mapping support

Generate Cpp Interface used in implementing mapping from one EXPRESS schema to another EXPRESS schema.

Menu path: Schemata>Generate Interface>Cpp 2010 mapping support

Source and target schema names must be specified. The actual Express schema must exist as a **dictionary model** in the *EDMdatabase*™ before this command can be successfully performed.

With option "source as interface to target" set, a C++ header file

Arguments:

Source schema:	Specify the name of the source schema in the *EDMdatabase*™ to generate a C++ early binding interface for. If entitites and defined types in source schema also exist in the target schema, it will be omitted in the generated header file. Schema names are case insensitive. When activating the **Select** button, the names of all existing schemata will be displayed in the related selection list.
Target schema:	Specify the name of the target schema in the *EDMdatabase*™. Schema names are case insensitive. When activating the **Select** button, the names of all existing schemata will be displayed in the related selection list.
Mapping model repository	Specify the name of the data repository to search for the mapping model. Repository names are case sensitive. When activating the **Select** button, the name of all repositories will be displayed in the related selection list.
Mapping model	Specify the name of the mapping model. Model names are case sensitive. The <Source Model> can be open or closed. When activating the **Select** button, the name of all models located in the repository specified in the <Mapping model repository> will be displayed in the related selection list.
Cpp interface output directory:	Location for storing of C++ source. When activating the **Select** button, a window is opened to allow browsing the directory structure and select the desired directory.
Namespace:	Unique name for namespace (if omitted - name of schema will be used instead)

Options:

accumulating command output:	A global option that appends the command output to the *EDMsupervisor*™ output window. Otherwise the output window is refreshed and only the last command output is displayed.