Dynamic management of Any values 7

An **any** can be passed to a program that doesn't have any static information for the type of the **any** (code generated for the type by an IDL compiler has not been compiled with the object implementation). As a result, the object receiving the **any** does not have a portable method of using it.

The facility presented here enables traversal of the data value associated with an **any** at runtime and extraction of the primitive constituents of the data value. This is especially helpful for writing powerful generic servers (bridges, event channels supporting filtering, etc.).

Similarly, this facility enables the construction of an **any** at runtime, without having static knowledge of its type. This is especially helpful for writing generic clients (bridges, browsers, debuggers, user interface tools, etc.).

Contents

This chapter contains the following sections.

7.1 Overview

Any values can be dynamically interpreted (traversed) and constructed through **DynAny** objects. A **DynAny** object is associated with a data value which may correspond to a copy of the value inserted into an **any**. The **DynAny** object may be seen as owning a pointer to an external buffer which holds some representation of the data value.

A constructed **DynAny** object is a **DynAny** object associated with a constructed type. There is a different interface, inheriting from the **DynAny** interface, associated with each kind of constructed type in IDL (struct, sequence, union, or array). A constructed **DynAny** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a component of the constructed data value.

As an example, a **DynStruct** is associated with a struct value. This means that the object may be seen as owning a pointer to a external buffer which holds a representation of struct. The **DynStruct** object exports operations that enable the creation of new **DynAny** objects, each of them associated with a member of the struct.

If a **DynAny** object has been created from another (a constructed) **DynAny** object then the buffer pointed to by the first **DynAny** object is logically contained within the buffer pointed by the second **DynAny** object.

Destroying a **DynAny** object implies deleting the buffer it points to and also destroying all **DynAny** objects obtained from it. Invoking operations using references to descendants of a destroyed **DynAny** object leads to unpredictable results. Note that releasing a reference to a **DynAny** object will not delete the buffer pointed by the object, since the object indeed exists (it has not been explicitly destroyed).

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, then he or she should first create a **DynAny** for the component and, after that, make a copy of the created **DynAny** object.

The behavior of **DynAny** objects has been defined in order to enable efficient implementations in terms of allocated memory space and speed of access. **DynAny** objects are intended to be used for traversing values extracted from **any**s or constructing values of **any**s at runtime. Their use for other purposes is not recommended.

7.2 DynAny API

The **DynAny** API comprises the following IDL definitions to be included in the CORBA module:

```
// IDL
interface DynAny {
    exception Invalid {};
    exception InvalidValue {};
    exception TypeMismatch {};
    exception InvalidSeq {};

    typedef sequence<octet> OctetSeq;
    TypeCode type ();

    void assign (in DynAny dyn_any) raises (Invalid);
    void from_any (in any value) raises (Invalid);
    any to_any() raises (Invalid);

    void destroy();

    DynAny copy();

    void insert_boolean(in boolean value) raises (InvalidValue);
    void insert_octet(in octet value) raises (InvalidValue);
    void insert_char(in char value) raises (InvalidValue);
    void insert_short(in short value) raises (InvalidValue);
    void insert_ushort(in unsigned short value) raises (InvalidValue);
    void insert_long(in long value) raises (InvalidValue);
    void insert_ulong(in unsigned long value) raises (InvalidValue);
    void insert_float(in float value) raises (InvalidValue);
    void insert_double(in double value) raises (InvalidValue);
    void insert_string(in string value) raises (InvalidValue);
    void insert_reference (in Object value) raises (InvalidValue);
    void insert_typecode (in TypeCode value) raises (InvalidValue);
    void insert_longlong(in long long value) raises(InvalidValue);
    void insert_ulonglong(in unsigned long long value) raises(InvalidValue);
    void insert_longdouble(in long double value) raises(InvalidValue);
    void insert_wchar(in wchar value) raises(InvalidValue);
    void insert_wstring(in wstring value) raises(InvalidValue);
    void insert_any(in any value) raises(InvalidValue);

    boolean get_boolean() raises (TypeMismatch);
    octet get_octet() raises (TypeMismatch);
    char get_char() raises (TypeMismatch);
    short get_short() raises (TypeMismatch);
    unsigned short get_ushort () raises (TypeMismatch);
    long get_long() raises (TypeMismatch);
    unsigned long get_ulong() raises (TypeMismatch);
    float get_float() raises (TypeMismatch);
```

```
        double get_double() raises (TypeMismatch);
        string get_string() raises (TypeMismatch);
        Object get_reference() raises (TypeMismatch);
        TypeCode get_typecode () raises (TypeMismatch);
        long long get_longlong() raises(TypeMismatch);
        unsigned long long get_ulonglong() raises(TypeMismatch);
        long double get_longdouble() raises(TypeMismatch);
        wchar get_wchar() raises(TypeMismatch);
        wstring get_wstring() raises(TypeMismatch);
        any get_any() raises (TypeMismatch);

        DynAny current_component ();
        boolean next ();
        boolean seek (in long index);
        void rewind ();
};

interface DynFixed : DynAny {
        OctetSeq get_value ();
        void set_value (in OctetSeq val) raises (InvalidValue);
};

interface DynEnum: DynAny {
        attribute string value_as_string;
        attribute unsigned long value_as_ulong;
};

typedef string FieldName;

struct NameValuePair {
        FieldName id;
        any value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct: DynAny {
        FieldName current_member_name ();
        TCKind current_member_kind ();
        NameValuePairSeq get_members();
        void set_members(in NameValuePairSeq value)
                raises (InvalidSeq);
};
```

```
interface DynUnion: DynAny {
    attribute boolean set_as_default;
    DynAny discriminator ();
    TCKind discriminator_kind ();
    DynAny member ();
    attribute FieldName member_name;
    TCKind member_kind ();
};

typedef sequence<any> AnySeq;

interface DynSequence: DynAny {
    attribute unsigned long length;
    AnySeq get_elements ();
    void set_elements (in AnySeq value)
            raises (InvalidSeq);
};

interface DynArray: DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

7.2.1 Locality and usage constraints

DynAny objects are intended to be local to the process in which they are created and used. This means that references to **DynAny** objects cannot be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a **MARSHAL** system exception.

Since their interfaces are specified in IDL, **DynAny** objects export operations defined in the standard **CORBA::Object** interface. However, any attempt to invoke operations exported through the **Object** interface may raise the standard **NO_IMPLEMENT** exception.

An attempt to use a **DynAny** object with the DII may raise the **NO_IMPLEMENT** exception.

7.2.2 Creating a DynAny object

A **DynAny** object can be created as a result of:

- invoking an operation on an existing **DynAny** object
- invoking an operation exported by the ORB

Actually, a constructed **DynAny** object support operations that enable the creation of new **DynAny** objects encapsulating access to the value of some constituent. **DynAny** objects also support the **copy** operation for creating new **DynAny** objects.

In addition, the ORB can act as a factory of **DynAny** objects in the same way as with **TypeCode** objects. Therefore, the standard **ORB** interface includes the following operations:

```
interface ORB {
    ...
    DynAny create_dyn_any (in any value);
    DynAny create_basic_dyn_any(in TypeCode type)
                raises (InconsistentTypeCode);
    DynStruct create_dyn_struct(in TypeCode type)
                raises (InconsistentTypeCode);
    DynSequence create_dyn_sequence(in TypeCode type)
                raises (InconsistentTypeCode);
    DynArray create_dyn_array(in TypeCode type)
                raises (InconsistentTypeCode);
    DynUnion create_dyn_union(in TypeCode type)
                raises (InconsistentTypeCode);
    DynEnum create_dyn_enum(in TypeCode type)
                raises (InconsistentTypeCode);
    DynFixed create_dyn_fixed(in TypeCode type)
                raises (InconsistentTypeCode);
    ...
};
```

The **create_dyn_any** operation creates a new **DynAny** object from an **any** value. A duplicate of the **TypeCode** associated with the **any** value is assigned to the resulting **DynAny** object. The value associated with the **DynAny** object is a copy of the value in the original any.

The rest of the operations used to create **DynAny** objects receive a **TypeCode** input parameter and throw the **InconsistentTypeCode** exception if the **TypeCode** passed as a parameter is not consistent with the operation.

Dynamic interpretation of an **any** usually involves creating a **DynAny** object using **create_dyn_any** as the first step. Depending on the type of the **any**, the resulting **DynAny** object reference can be narrowed to a **DynStruct**, **DynSequence**, **DynArray**, **DynUnion** or **DynEnum** object reference.

Dynamic creation of an **any** containing a struct data value typically involves creating a **DynStruct** object using **create_dyn_struct**, passing the **TypeCode** associated with the struct data value to be created. Then, components of the struct can be initialized by means of invoking operations on the resulting **DynStruct** object or **DynAny** objects generated for each member of the struct. Finally, once the data value pointed by the **DynStruct** object has been properly initialized, the **to_any** operation can be invoked. The same approach would be followed for dynamic creation of sequences, unions, etc.

Dynamic creation of an **any** containing a value of a basic data type typically involves creating a **DynAny** object using **create_basic_dyn_any**, passing the **TypeCode** associated with the basic data type value to be created. Then, the value can be

initialized by means of invoking operations on the resulting **DynAny** object (**insert_boolean** if the **DynAny** is of type **boolean**, etc.). Finally, the **to_any** operation can be invoked.

7.2.3 The DynAny interface

The following operations can be applied to a **DynAny** object:

- Obtaining the **TypeCode** associated with the **DynAny** object
- Generating an **any** value from the **DynAny** object
- Destroying the **DynAny** object
- Creating a **DynAny** object as copy of the **DynAny** object
- Inserting/getting a value of some basic type into/from the **DynAny** object
- Iterating through the components of a **DynAny**
- Obtaining the TypeCode associated to the DynAny object
- Initializing a **DynAny** object from another **DynAny** object
- Initializing a **DynAny** object from an **any** value
- Generating an **any** value from the **DynAny** object
- Destroying the **DynAny** object
- Creating a **DynAny** object as copy of the **DynAny** object
- Inserting/Getting a value of some basic type into/from the **DynAny** object
- Iterating through the components of a **DynAny**

Obtaining the TypeCode associated with a DynAny object

A **DynAny** object is created with a **TypeCode** value assigned to it. This **TypeCode** value determines the type of the value handled through the **DynAny** object. The **type** operation returns the **TypeCode** associated with a **DynAny** object:

```
TypeCode type();
```

Note that the **TypeCode** associated with a **DynAny** object is initialized at the time the **DynAny** is created and cannot be changed during lifetime of the **DynAny** object.

Initializing a DynAny object from another DynAny object

The **assign** operation initializes the value associated to a **DynAny** object with the value associated to another **DynAny** object:

```
void assign(in DynAny dyn_any) raises(Invalid);
```

If an invalid **DynAny** object is passed (it has a different typecode or doesn't contain a meaningful value), the **Invalid** exception is returned.

Initializing a DynAny object from an any value

The **from_any** operation initializes the value associated to a **DynAny** object with the value contained in an **any**:

void from_any(in any value) raises(Invalid);

If an invalid **any** is passed (it has a different typecode or hasn't been assigned a value) the **Invalid** exception is returned.

Generating an any value from a DynAny object

The **to_any** operation creates an **any** value from a **DynAny** object:

any to_any() raises(Invalid);

If the **DynAny** object has not been correctly created or doesn't contain a meaningful value (it hasn't been properly initialized, for example), the **Invalid** exception is returned.

A duplicate of the **TypeCode** associated with the **DynAny** object is assigned to the resulting **any**. The value associated with the **DynAny** object is copied into the **any**.

Destroying a DynAny object

The **destroy** operation destroys a **DynAny** object. This operation frees any resources used to represent the data value associated with a **DynAny** object.

void destroy();

Destruction of a **DynAny** object implies destruction of all **DynAny** objects obtained from it.

Destruction of **DynAny** objects should be handle with care taking into account issues dealing with representation of data values associated with **DynAny** objects.

If the programmer wants to destroy a **DynAny** object but still wants to manipulate some component of the data value associated with it, he or she should first create a **DynAny** for the component and then make a copy of the created **DynAny** object.

Creating a copy of a DynAny object

The **copy** operation enables the creation of a new **DynAny** object whose value is a deep copy of the value pointed by the **DynAny** object:

DynAny copy();

Accessing a value of some basic type in a DynAny object

The insert and get operations have been defined to enable insertion/extraction of basic data type values into/from a **DynAny** object.

Insert operations raise the **InvalidValue** exception if the value inserted is not consistent with the type of the accessed component in the **DynAny** object.

Get operations raise the **TypeMismatch** exception if the accessed component in the **DynAny** is of a type that is not consistent with the requested type.

These operations are necessary to handle basic **DynAny** objects but are also helpful to handle constructed **DynAny** objects. Inserting a basic data type value into a constructed **DynAny** object implies initializing the next component of the constructed data value associated with the **DynAny** object. For example, invoking **insert_boolean** in a **DynStruct** implies inserting a boolean data value as the next member of the associated struct data value.

In addition, availability of these operations enable the traversal of **any**s associated with sequences of basic data types without the need to generate a **DynAny** object for each element in the sequence.

Iterating through components of a DynAny

The **DynAny** interface allows a client to iterate through the components of the struct data value pointed by a **DynStruct** object.

As mentioned above, a **DynAny** object may be seen as owning a pointer to an external buffer that holds some representation of a data value. In addition, the **DynAny** object holds a pointer to a buffer offset where the current component is being represented.

The buffer pointer effectively points to the space used to represent the first component of the data value when the programmer creates the **DynAny** object. It also points to the first component each time **rewind** is invoked.

void rewind();

The **next** operation logically advances the pointer and returns TRUE if the resulting pointer points to a component, or FALSE if there are no more components. Invoking **next** on a **DynAny** associated with a basic data type value is allowed, but it always returns FALSE.

boolean next();

The programmer is able to inspect/initialize the component of the data value associated with the **DynAny** object by means of invoking **current_component** at each step during the iteration.

```
DynAny current_component();
```

The resulting **DynAny** object reference would be used to get/set the value of the component currently accessed. In order to get access to specific operations, the resulting **DynAny** object reference may be narrowed based on its **TypeCode**.

In order to construct an **any** associated with a sequence data value, for example, the programmer may first create the **DynAny** object invoking **create_dyn_sequence**. After doing so, the programmer may iterate through the elements of the sequence. At each step, an element in the sequence would be initialized by means of invoking **current_component** and using the returned **DynAny**. After that, **next** will be invoked. The end of the initialization process would be detected when **next** returns FALSE. At that point, the programmer would invoke **to_any** to create an **any**.

Operation **seek** logically sets a new offset for this pointer, returning TRUE if the resulting pointer points to a component or FALSE if there is no component at the designated offset. Invoking **seek** on a **DynAny** associated to a basic data type value is allowed but it only returns TRUE if the value passed as argument equals to zero.

```
boolean seek(in long index);
```

7.2.4 The DynFixed interface

DynFixed objects are associated with values of the IDL **fixed** type.

```
typedef sequence<octet> OctetSeq;
interface DynFixed : DynAny {
    OctetSeq get_value ();
    void set_value (in OctetSeq val) raises (InvalidValue);
};
```

The **get_value** operation returns the value of the **DynFixed** as a sequence of octet. Each octet contains either one or two decimal digits. If the fixed type has an odd number of decimal digits (which can be determined from the **TypeCode::fixed_digits** operation), then the representation begins with the first (most significant) digit. Otherwise, the first half-octet is all zero, and the first digit is in the second half-octet. The sign of the value, which is stored in the last half-octet of the sequence, shall be 0xD for negative numbers and 0xC for positive and zero values.

The **set_value** operation sets the value of the **DynFixed** with an **OctetSeq** having the same format as that described above. If the **OctetSeq** does not conform to the expected number of digits as determined by the **TypeCode**, the **InvalidValue** exception is raised.

7.2.5 The DynEnum interface

DynEnum objects are associated with enumerated values.

```
interface DynEnum: DynAny {
    attribute string value_as_string;
    attribute unsigned long value_as_ulong;
};
```

The **DynEnum** interface consists of two attributes: the **value_as_string** attribute which contains the value of the enum value as a string and the **value_as_ulong** which contains the value of the enum value as an unsigned long:

```
attribute string value_as_string;
attribute unsigned long value_as_ulong;
```

7.2.6 The DynStruct interface

DynStruct objects are associated with struct values and exception values.

```
typedef string FieldName;

struct NameValuePair {
    FieldName id;
    any value;
};

typedef sequence<NameValuePair> NameValuePairSeq;

interface DynStruct: DynAny {
    FieldName current_member_name ();
    TCKind current_member_kind ();
    NameValuePairSeq get_members();
    void set_members(in NameValuePairSeq value)
            raises (InvalidSeq);
};
```

The **current_member_name** operation returns the name of the member currently being accessed.

```
FieldName current_member_name ();
```

This operation may return an empty string since the **TypeCode** of the struct being manipulated may not contain the names of members in the struct.

current_member_kind returns the TCKind associated with the current member being accessed.

```
TCKind current_member_kind ();
```

It is possible to obtain a sequence of name/value pairs describing the name and the value of each member in the struct associated with a **DynStruct** object using the **get_members** operation:

```
NameValuePairSeq get_members();
```

The **set_members** operation initializes the struct data value associated with a **DynStruct** object from a sequence of name value pairs:

```
void set_members(in NameValuePairSeq value)
        raises (InvalidSeq);
```

Members must appear in the **NameValuePairSeq** in the order in which they appear in the IDL specification of the struct. This operation raises the **InvalidSeq** exception if an inconsistent name or value is passed as argument (for example, the **NameValuePairSeq** does not match the members of the struct, it's too long/short, or member values are passed in the wrong order).

DynStruct objects can also be used for handling exception values. In that case, members of the exceptions are handled in the same way as members of a struct.

7.2.7 The DynUnion interface

DynUnion objects are associated with unions.

```
interface DynUnion: DynAny {
    attribute boolean set_as_default;
    DynAny discriminator ();
    TCKind discriminator_kind ();
    DynAny member ();
    attribute FieldName member_name;
    TCKind member_kind ();
};
```

The **DynUnion** interface allows for the insertion/extraction of an OMG IDL union type into/from a **DynUnion** object.

The discriminator operation returns a **DynAny** object reference that must be narrowed to the type of the discriminator in order to insert/get the discriminator value:

```
DynAny discriminator ();
```

Note that the type of the discriminator is contained in the **TypeCode** of the union.

The **member** operation returns a **DynAny** object reference that is used in order to insert/get the member of the union:

```
DynAny member ();
```

discriminator_kind and **member_kind** return the TCKind associated with the discriminator and member of the union, respectively:

```
TCKind discriminator_kind ();
TCKind member_kind ();
```

The **member_name** attribute allows for the inspection/assignment of the name of the union member without checking the value of the discriminator.

The **set_as_default** attribute determines whether the discriminator associated with the union has been assigned a valid default value.

Union values can be traversed using the operations defined in "Iterating through components of a DynAny" on page 7-9. In that case, the first component in the union corresponds to the discriminator while the second corresponds to the actual value of the union. Operation **next** should then be called twice.

7.2.8 The DynSequence interface

DynSequence objects are associated with sequences.

```
typedef sequence<any> AnySeq;

interface DynSequence: DynAny {
    attribute unsigned long length;
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

The **length** attribute contains the number of elements contained in (or to be contained in) the sequence; its value is initalized to zero for unbounded sequences:

```
attribute unsigned long length;
```

The **get_elements** and **set_elements** operations return and receive respectively a sequence of **any**s containing each of the elements of the sequence:

```
AnySeq get_elements();
void set_elements(in AnySeq value);
```

The **set_elements** operation raises the **InvalidSeq** exception if an inconsistent value is passed in the sequence of **any** values passed as argument (for example, the **AnySeq** is too long/short).

7.2.9 The DynArray interface

DynArray objects are associated with arrays.

```
interface DynArray: DynAny {
    AnySeq get_elements();
    void set_elements(in AnySeq value)
            raises (InvalidSeq);
};
```

The **get_elements** and **set_elements** operations return and receive respectively a sequence of **any**s containing each of the elements of the array:

```
AnySeq get_elements();
void set_elements(in AnySeq value);
```

The **set_elements** operation raises the **InvalidSeq** exception if an inconsistent value is passed in the sequence of **any** values passed as argument (for example, the **AnySeq** is too long/short).

Note that the dimension of the array is contained in the **TypeCode** which is accessible through the **type** attribute.

7.3 Usage in C++ language

7.3.1 Dynamic creation of CORBA::Any values

Creating an any which contains a struct

Consider the following IDL definition:

```
// IDL
struct MyStruct {
    long member1;
    boolean member2;
};
```

The following example illustrates how a **CORBA::Any** value may be constructed on the fly containing a value of type **MyStruct**:

```
// C++
CORBA::ORB_var orb;
CORBA::StructMemberSeq mems(2);
CORBA::Any result;
long value1;
boolean value2;

mems[0].name = CORBA::string_dup("member1");
mems[1].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
mems[0].name = CORBA::string_dup("member2");
mems[1].type =
   CORBA::TypeCode::_duplicate(CORBA::_tc_boolean);

CORBA::TypeCode_var new_tc = orb->create_struct_tc (
                               "IDL:MyStruct:1.0",
                               "MyStruct",
                               mems
                             );

// construct the DynStruct object. Values for members have
// read in the value1 and value2 variables

DynStruct_ptr dyn_struct = orb->create_dyn_struct (new_tc);
dyn_struct->insert_long(value1);
dyn_struct->insert_boolean(value2);
result = dyn_struct->to_any();
dyn_struct->destroy ();
CORBA::release(dyn_struct);
```

7.3.2 Dynamic interpretation of CORBA::Any values

Filtering of events

Suppose there is a CORBA object which receives events and prints all those events which correspond to a data structure containing a member called **`is_urgent`** whose value is TRUE.

The following fragment of code corresponds to a method which determines if an event should be printed or not. Note that the program allows several struct events to be filtered with respect to some common member.

```
// C++
CORBA::Boolean Tester::eval_filter(const CORBA::Any &event)
{
   CORBA::Boolean success = FALSE;

   // First, typecode is extracted from the event.  This
   // is necessary to get struct member names:
   CORBA::TypeCode_var event_type = event->type();

   // The filter only returns true if the event is a struct:
   if (event_type->kind() == CORBA::tk_struct)
   {
      DynAny_ptr dyn_any = orb->create_dyn_any(event);
      DynStruct_ptr dyn_struct= DynStruct::_narrow(dyn_any);
      CORBA::release(dyn_any);

      CORBA::Boolean found = FALSE;

      do
      {
         CORBA::String_var member_name =
                     dyn_struct->current_member_name();

         found = (strcmp(member_name, "is_urgent") == 0);
      } while (!found && !dyn_struct->next());


      if (found)
      {
         // We only create a DynAny object for the member
         // we were looking for:
         CORBA::DynAny_var dyn_member =
                     dyn_struct->current_component ();
         success = dyn_member->get_boolean();
      };

      dyn_struct->destroy();
      CORBA::release(dyn_struct);
   };

   return success;
};
```