OSF DCE SIG
Request For Comments: 22.0

N. Kincl (HP)
D. Thompson (HP)
R. Webber (HP)
January 1993

ADAPTING DME TO MANAGE DCE-BASED SERVICES

1. INTRODUCTION

The DME model provides a structure to allow for the management of systems and networks. This paper discusses how the DME model could be used and extended to efficiently manage DCE-based services.

This is one in a series of position papers relating Hewlett-Packard's understanding of issues associated with the integration of DME technologies. The other papers include:

(a) *Interface Modeling and Implementation within DME* [KT1].

One of the basic premises of the DME evaluation team in making their technology choices is that multiple object-spaces are a reality of the environment. Any viable management solution must deal with this is some rational way. This paper looks at the primary models that are implemented in the technologies selected by OSF as part of DME and presents a rational view of how they interrelate. This is accomplished by relating these models to the reference model for management defined by X/Open and the OMG object model. This paper also considers how the choice of an interface definition language (IDL) affects the interoperation of the various object spaces. It evaluates the alternatives available to DME for Interface Definition and makes a specific recommendation.

(b) *Use of OMG-IDL for System Management*.

This document presents a set of stylized use and extensions to OMG-IDL to support building proxies for ISO managed objects.

This paper assumes a basic understanding of ASN.1, GDMO and DCE RPC.

2. DCE AND THE DME MODEL

Managing DCE services[1] using DME should be of particular interest to OSF and its member companies. The DME announcements have addressed issues involved in DME using DCE. However, little has yet been said on how DME will manage DCE services. In particular, there needs to be access from the DME object world to the management interface of the DCE server. The DME architecture provides for this access through the adaptor objects. However, neither the DME nor the DCE architectures have yet defined how the DME adaptor objects will communicate with DCE servers.

2.1. Consistent Interface Definition

What is needed is a consistent way of defining the management interface of DCE services. Unfortunately, there are two conflicting areas across which we would like to apply consistency:

(a) First, we would like to have consistency across both the management and functional interfaces of a server. The interface definition language (IDL) used for the functional interface is DCE IDL. To provide the consistency here, DCE IDL would be used to define the management interface.

(b) Second, we would like to maintain consistency across management interfaces of different resources. It is desirable to provide a consistent syntax and semantics for the management functions of not just the DCE

---

1. We use ''DCE service'' to mean both the services that comprise DCE as well as services built using DCE (DCE-based service).

services but also other network, system and application resources. In particular, there is a desire to provide consistency with the various standards bodies. To provide the consistency here, Guidelines for the Definition of Managed Objects (GDMO)[2] would be used to define the management interface.

Clearly it is not possible to meet both of these desires. We have chosen to opt for consistency across management interfaces of different types of resources. This adds a slight burden to the designer of the DCE service. However, it greatly simplifies the ability to manage a diverse set of resource in a consistent manner. This also simplifies the ability to provide a standards-based management solution for DCE services.

2.2. Consistent Implementation

It is not sufficient to have a consistent way of defining the management interface. Interoperability also requires a consistent way of implementing the management interface. For example, the OSI standards use CMIS/CMIP as the implementation of the GDMO-defined interface. For DCE, we would like to maintain the efficiency of the DCE distribution mechanisms. DCE services should not have to implement yet another networking protocol stack, or increase overhead by always forcing management interactions to go through a separate management agent. The model provides for a method of direct management using RPC.

There are two ways of providing a consistent way of implementing the managed objects within the service:

(a) *Separate agent.*

A management agent that is separate from the service can be used. All requests from the management applications go to the agent. The agent then uses a private protocol between itself and the service.

(b) *Direct implementation.*

The service implements the management interface directly. The service exports an RPC-based Management Interface. It acts as its own management agent, dealing with requests to all of the Agent Functions directly.

Providing a separate agent merely pushes the consistency problem to the agent/service interface. This is provided at the cost of requiring all management interactions to go through the agent. Defining a direct interface still allows for an agent to be configured, if necessary. However, it does not require an agent and allows a management application to bind directly to the interface.

A direct interface requires a way of mapping operations on a Managed Object into calls to an interface of the service. Two mappings are needed to translate the GDMO definitions into calls to an interface. First, the ASN.1 type definitions need to be mapped into DCE IDL type definitions. Second, the management operations defined and implied by the GDMO must be mapped into DCE RPC calls. This second mapping includes more than just access to the attributes or execution of the actions defined in the GDMO. It requires support for management functionality such as dealing with agents, scoping and filtering. This paper addresses dealing with agents. It touches on the scoping and filtering. We will deal with naming in a future position paper.

In general, mapping between IDL definitions and protocols is problematic for a number of reasons. For example, mapping and translation between CMIP and SNMP in not generally viable. However, due to the highly procedure-oriented nature of DCE, the mapping from GDMO to DCE IDL has proven to be relatively straight forward and reasonable. Hewlett-Packard's Information Architecture Group[3] has demonstrated this mapping in prototypes. Our demonstrations include compiler technology that creates the DCE IDL definition and associated convenience routines from the GDMO.

A detailed description of the model can be found in [KT1]. Applying the model to DCE, the Agent Functions (and thus the implementation of the Managed Objects) become part of the Managed Resource. DCE RPC is used as the transport protocol.

---

2. Except for the SNMP community, most standards work is converging on using GDMO to define managed objects. See [KT1] for a discussion about the advantages and limitations of various definition languages.

3. IAG has been restructured into the Networked Systems Architecture function.

It would be theoretically possible to create XMP access to the RPC protocol. It would require a metadata repository and support for interpretive stubs[4] in DCE. We do not feel that providing access to the RPC interface from XMP is a good idea and mention it only for the sake of completeness. It introduces significant complexity for both the programmer as well as the platform and provides minimal benefits.

3. MAPPING ASN.1 TO DCE DATA TYPES

This is a proposed mapping from Abstract Syntax Notation One (ASN.1) [X.208] to the DCE RPC Interface Definition Language (IDL) [AppDevGd]. It should be pointed out that the mapping is not a perfect mapping. ASN.1 is an abstract notation, which we are mapping to a concrete representation essentially the same as in the C language. Whereas ASN.1 has little or no limitations (for example, integers can be an arbitrary size), DCE IDL does have specific limits. However, DCE IDL is similar to the actual languages that will be used to program these resources, so these limitations are acceptable.

Throughout this paper, we have used ‘‘DCE IDL’’ to refer to DCE RPC IDL. This avoids confusion with the meaning of IDL as either a generic Interface Definition Language or the Object Management Group’s IDL.

3.1. Basic Data Types

3.1.1. `BOOLEAN`

The ASN.1 `BOOLEAN` type is mapped to the DCE IDL type `boolean`.

3.1.2. `INTEGER`

The ASN.1 `INTEGER` type is mapped to the DCE IDL integer types. The default mapping for an `INTEGER` with no subtyping is `long int`. The following table gives the mapping for other `INTEGER` subtypes. Integers that do not fit into a `hyper` are not supported.

```
         min                  max            DCE IDL Type
  -------------------  -------------------  --------------
      0 <= min < 2^8        0 <= max < 2^8   unsigned small
      0 <= min < 2^16     2^8 <= max < 2^16  unsigned short
      0 <= min < 2^32    2^16 <= max < 2^32  unsigned long
      0 <= min < 2^64    2^32 <= max < 2^64  unsigned hyper
   -2^7 <= min < 0       -2^7 <= max < 2^7   small
  -2^15 <= min < 0      -2^15 <= max < 2^15  short
  -2^31 <= min < 0      -2^31 <= max < 2^31  long
  -2^63 <= min < 0      -2^63 <= max < 2^63  hyper
```

3.1.3. `REAL`

The ASN.1 `REAL` type is mapped to the DCE IDL type `double`. Since ASN.1 does not specify precision for `REAL`, `double` was chosen in preference to float to allow the highest precision possible.

3.1.4. `ENUMERATED`

The ASN.1 type `ENUMERATED` is mapped to the most constrained DCE IDL integer that can represent all of the enumerated values. The DCE `enum` is not used since it does not allow you to specify the values of the enumeration constants. It is recommended that `const` definitions be used to represent each of the enumerated values. For example:

```
EnumEg ::= ENUMERATED {a(-2), b(34), c(380)}
```

maps to:

---

4. Interpretive marshalling as well as interpretive generation of packets.

```
typedef short EnumEg;
const short EnumEg_a = −2;
const short EnumEg_b = 34;
const short EnumEg_c = 380;
```

The limit on enumerated values is the same as the limit on integers. See Section 3.3.1 ‘‘Field names’’ for the significance of the names used in the definitions.

3.1.5. BIT STRING

There is no direct mapping from the ASN.1 `BIT STRING` type to a base DCE IDL type. Consequently a DCE IDL type must be constructed to represent bit strings. Both definite and indefinite length `BIT STRING`s are mapped to:

```
struct {
        unsigned long length_in_bytes;
        [size_is(length_in_bytes), ptr] byte *bit_string;
}
```

The `length_in_bytes` field represents the number of bytes pointed to by `bit_string`. The first byte in this array is not part of the bit string (but is included in the length). It has a value between 0 and 7, indicating the number of unused bits in the final byte. If the bit string is empty, the array shall consist of a single byte set to zero. This representation is consistent with the Basic Encoding Rules (BER) [X.209].

3.1.6. OCTET STRING

The ASN.1 type `OCTET STRING` has two different DCE IDL byte array representations according to whether or not it is of fixed length. In the fixed length case `OCTET STRING (SIZE(`*n*`))` the mapping is `byte` *<name>*`[`*n*`]`.

In the case of an indefinite length definition (no `SIZE` subtype) the mapping is:

```
struct {
        unsigned long size;
        [size_is(size), ptr] byte *o_str;
}
```

3.1.7. SEQUENCE

The ASN.1 type `SEQUENCE` is mapped to the DCE IDL type `struct`. Each element in the `SEQUENCE` has a corresponding field in the `struct`, the type being derived from the ASN.1 type (using the mapping described in this document) and the name derived from the ASN.1 identifier (if present). For an element that is defined as `OPTIONAL` a discriminated `union` is created. The discriminator is a `boolean`, where a `TRUE` value indicates the presence of the optional element. The contents of the discriminated `union` are either the element (with the appropriate type representation) or a dummy `byte`. For example the ASN.1 type:

```
SEQUENCE {
        temp            REAL OPTIONAL,
        age             INTEGER (0..120),
        alive           BOOLEAN
}
```

is mapped to the DCE IDL data type:

```
union temp_type switch(boolean) {
        case TRUE:      float value;
        default:        byte dummy;
};
```

```
struct {
        temp_type       temp;
        unsigned small  age;
        boolean         alive;
}
```

If there is no name given for a particular element of the `SEQUENCE` then a field name will have to be generated that is unique within the DCE IDL `struct` (see Section 3.3.1 ‘‘Field names’’). Note that there is no DCE IDL support for `DEFAULT` values. As in ASN.1, support needs to be provided at run-time. Support could be provided through convenience routines, but the IDL treats them the same as `OPTIONAL`.

3.1.8. `SEQUENCE OF`

The ASN.1 `SEQUENCE OF` type may be of definite or indefinite length. The definite length form `SEQUENCE SIZE (`*n*`) OF` is mapped to a DCE IDL array `<`*type*`> <`*name*`>[`*n*`]`.

For example:

```
Example ::= SEQUENCE SIZE (21) OF INTEGER (-1..27)
```

maps to:

```
small Example[21]
```

The indefinite length case (i.e., `SEQUENCE OF`) maps to:

```
struct {
        unsigned long size;
        [size_is(size), ptr] <type> *seq;
}
```

3.1.9. `SET`

Like `SEQUENCE`, the ASN.1 `SET` type is also mapped to a `struct` in DCE IDL. Thus:

```
SET {
        temp            REAL,
        age             INTEGER (0..120),
        alive           BOOLEAN
}
```

maps to:

```
struct {
        double temp;
        unsigned small age;
        boolean alive;
}
```

`OPTIONAL` elements of a `SET` are handled the same way as those of a `SEQUENCE`. See Section 3.1.7 ‘‘`SEQUENCE`’’ for details.

A feature of `SET`s that is not preserved in the mapping to DCE IDL is the insignificance of the ordering of elements. Since DCE IDL provides no representation that supports this feature, the ordering of the fields in the struct will be the same as the ordering of the elements in the `SET`. This causes a problem only if one considers the following two ASN.1 definitions to be identical:

(a) `SET { a REAL, b BOOLEAN }`

(b) `SET { b BOOLEAN, a REAL }`

It seems reasonable to treat the two as different ASN.1 specifications that may happen to have the same Basic Encoding Rules (BER) mapping. If treating these two definitions as different were to be considered a limitation, it would be possible to define a canonical ordering of fields within the DCE IDL `struct`.

3.1.10. `SET OF`

The ASN.1 `SET OF` type is mapped to DCE IDL in the same way as the `SEQUENCE OF` type. The only difference between the types is that there are no semantics associated with the order of the elements in a `SET OF` — a difference that cannot be reflected in the syntax. Thus the definite length form `SET SIZE (`*n*`) OF` is mapped to a DCE IDL array `<`*type*`> <`*name*`>[`*n*`]`.

For example:

```
Example ::= SET SIZE (21) OF INTEGER (-1..27)
```

maps to:

```
small Example[21]
```

The indefinite length case (i.e., `SET OF`) maps to:

```
struct {
        unsigned long size;
        [size_is(size), ptr] <type> *set;
}
```

3.1.11. `CHOICE`

DCE IDL has a data type that serves very well as a mapping for the ASN.1 type `CHOICE`, namely the `union switch`. The use of this mapping is best illustrated by an example. The ASN.1 type:

```
CHOICE {
        temp REAL,
        age INTEGER (0..120),
        alive BOOLEAN
}
```

maps to:

```
union switch (short id) {
        case 0:
                double temp;
        case 1:
                unsigned small age;
        case 2:
                boolean alive;
}
```

Note that the values for the `id` are monotonically increasing values beginning at zero. Note also that these are not related to any ASN.1 tag values that may be associated with any of the `CHOICE` elements. The ASN.1 tag values are used to distinguish the fields in BER. The case labels perform the same function in a simpler way.

3.1.12. `ANY`

The ASN.1 type `ANY` is has no obvious equivalent in DCE IDL, and for that reason no mapping is proposed. Note that use of `ANY` without the `DEFINED BY` alternative is deprecated in 1988 ASN.1 [X.208].

3.1.13. `OBJECT IDENTIFIER`

In a DCE based system there may be unique identifiers other than `OBJECT IDENTIFIER`s that are being passed around. For example there will often be a requirement to use DCE UUIDs instead. For that reason the proposed DCE IDL representation of an `OBJECT IDENTIFIER` is designed to be able to hold other unique identifiers. The structure is:

```
struct {
        uuid_t id_type;
        unsigned long size;
        [ptr, size_is(size)] byte *id;
}
```

The `id_type` field indicates which type of unique identifier is being represented. The type of this field is a `uuid_t` — that is a 128 bit DCE UUID. The purpose of this is to be able to define values that indicate well known id types such as OID or UUID. It also allows individuals to define their own types and be able to generate a value that is guaranteed not to have been previously allocated (by using the DCE `uuidgen` command). The size field contains the number of bytes in the id field. The id field is a pointer to a representation of the identifier that is particular to its type. A suggested representation of an OID is to use the BER to encode it. This has the problem that the program, rather than the stub, has to be able to perform the encoding and decoding. A more natural DCE IDL representation would be an array of `short int`, but transmitting this in an array of `byte` would create byte ordering problems.

3.1.14. Character string types

All the ASN.1 character string types (e.g., `NumericString`, `VisibleString`, `GeneralString`, etc.) of definite length are currently mapped to DCE IDL in the following way:

```
VisibleString (SIZE(n))
```

maps to:

```
[string] char <name>[n+1]
```

The DCE IDL type is one character longer than the ASN.1 type in order to leave room for a null terminator. Whereas the indefinite length versions map to the following DCE IDL type:

```
struct {
        unsigned long size;
        [size_is(size), string, ptr] char *string;
}
```

Note that in both cases the attribute string is applied to the `char` array — this indicates to DCE that there is a null-terminated string contained in it and only to bother transmitting data up to and including the null. This representation loses information on the type of the string, this information could be included by adding an extra type field to the structure. This is not proposed, however, since there is no accepted way of identifying string types within the C language.

3.2. Constructed Types

The previous section described how to map the basic ASN.1 types to DCE IDL. The basic types can be constructed into more complicated types. This section elaborates on constructed types. Consider the following ASN.1 type:

```
SEQUENCE
{
        CHOICE {
                a INTEGER (0..1000),
                b REAL,
                c SET SIZE (32) OF BOOLEAN
        } x,
        d VisibleString (SIZE (100))
}
```

This would map to:

```
struct {
        union switch (short id) {
                case 0:
                        unsigned short a;
                case 1:
                        double b;
                case 3:
                        boolean c[32];
        } x;
        [string] char d[101];
}
```

When defining ASN.1 it is necessary to define new types and then use those types in other definitions. This should be mapped to DCE IDL through the use of `typedef`. For example:

```
Age ::= INTEGER (0..120)
BloodPressure ::= SEQUENCE {
        systolic        INTEGER (0..400),
        diastolic       INTEGER (0..400)
}
Temperature ::= REAL

PatientRecord ::= SEQUENCE {
        age             Age,
        bp              BloodPressure,
        temp            Temperature
}
```

maps to:

```
typedef unsigned small Age;
typedef struct {
        unsigned short systolic;
        unsigned short diastolic;
} BloodPressure;
typedef double Temperature;

typedef struct {
        Age age;
        BloodPressure bp;
        Temperature temp;
} PatientRecord;
```

3.3. Implementation Notes

3.3.1. Field names

During the course of this proposal names have been given to fields of structures and types that are based on the names in the ASN.1 definitions. As in ASN.1, however, these names have no bearing on what is sent on the wire and hence do not have to be agreed upon in order to provide interoperability — rather it is a local matter for any implementation (e.g., of a compiler).

3.3.2. Stub generator support

If this mapping were used by a stub compiler that was generating DCE RPC run-time stubs (such as a GDMO compiler), it would be possible to generate run-time code in support of some of the ASN.1 definitions. For example code could be generated to enforce bounds checking on `INTEGER`s (rather than simply mapping them to the most constrained DCE IDL integer available). It would also be possible to set `DEFAULT` values as defined by the ASN.1 specification.

3.3.3. Pointers versus open arrays

In all the mappings proposed here, indefinite length arrays are represented as pointers rather than open arrays. Open arrays are not used even though they may seem the more logical representation. This is for two reasons. First, with the current implementation of the DCE RPC run-time library and stub compiler, open arrays cannot be used as `[out]` only parameters. Second, types containing open arrays must be passed by reference. This run-time behavior would place an unacceptable constraint on the use of the DCE IDL data types described here.

3.4. Summary of Data Type Mappings

The following table summarizes a mapping from ASN.1 to DCE IDL data types. Though other mappings may be possible, it is important that *some* mapping be selected as the standard.

ASN.1 Data Type	DCE IDL Data Type
BOOLEAN	boolean
INTEGER	long int /* default */
REAL	double
ENUMERATED	int /* of suitable size */

BIT STRING	struct {
	unsigned long
	length_in_bytes;
	[size_is(length_in_bytes),
	ptr] byte *bit_string;
	}

OCTET STRING (SIZE(n))	byte <name>[n]

OCTET STRING	struct {
	unsigned long size;
	[size_is(size), ptr]
	byte *o_str;
	}

SEQUENCE or SET	struct { /* ... */ }

SEQUENCE OF <type>
or
SET OF <type>
	struct {
	unsigned long size;
	[size_is(size), ptr]
	<type> *seq;
	}

SEQUENCE SIZE(n) OF <type> or SET SIZE(n) OF <type>	<type> <name>[n]
CHOICE	union switch (short id) { /* ... */ }

`ANY`	`/* not supported */`[5]

`OBJECT IDENTIFIER`	`struct {`
	`  uuid_t id_type;`
	`  unsigned long size;`
	`  [size_is(size), ptr]`
	`    byte *id;`
	`}`[6]

`VisibleString (SIZE(n))`	`[string] char <name>[n+1]`

`VisibleString`	`struct {`
	`  unsigned long size;`
	`  [size_is(size), ptr,`
	`    string] char *string;`
	`}`

4. MAPPING GDMO TO DCE RPC

Having mapped the ASN.1 data types to DCE data types it is possible to map the GDMO templates to a set of DCE RPC definitions. The approach is to define an RPC call for each of the operations that could be performed given a set of GDMO templates. The model is that management applications wanting to perform an operation (such as set an attribute or invoke an action) would bind to the resource implementing the Managed Object and use the appropriate RPC. The resource may not be accessible directly by the application, in which case it may be accessed via a management agent (as in the OSI model). We also need to provide support for the basic functionality required by management. This includes support for agents and for scoping and filtering of requests.

4.1. Agent Support

With RPC, you always bind directly to the resource which will process your requests. Unfortunately, this is not an acceptable limitation of management. Often times the resource that needs to be managed needs to be accessed through an agent. Typically transient objects[7] will be accessed through an agent. For example, it is unrealistic to expect every process on a system to export an RPC interface. Rather, the system would have an agent that handled the Management Interface for all the processes on the system.

The agent must export the appropriate RPC interface. The agent distinguishes between requests made for the different managed objects it represents by means of an instance parameter. If the value of the instance parameter is null, then the RPC is directed at the bound object itself. Otherwise, the instance name will contain a name of the instance desired. The name could be either relative to the agent or a global name.

4.2. Scoping and Filtering

System management often requires the ability to scope or filter an operation over many resources. Scoping is a way of selecting managed objects based on criteria external to the object. An example of scoping is the CMIS scoping. CMIS allows a management application to direct a request at objects based on their level in the containment tree.[8] A more general scoping method could scope a request based on an arbitrary relationship external to the managed objects (such as ‘‘all objects that are used by my mission-critical applications’’).

---

5. No obvious equivalent. The use of `ANY` without the `DEFINED BY` is deprecated in 1988 ASN.1 [X.208].

6. This allows unique identifiers other than an ASN.1 `OBJECT IDENTIFIER` to be represented.

7. We plan to address static and transient objects further in a future paper.

8. Given a base object, the scope can be the base object alone, the nth level subordinates of the base object, the base object and all of its subordinates down to and including the nth level, or the base object and all of its subordinates.

Filtering allows for selection of the managed objects that should perform a particular management interaction to be based upon attributes of the managed object. All objects within the defined scope receive the request. Only those objects (or their agents) whose filter information matches the request will process the request.

Scoping and filtering allows only one operation to be issued, but executed by many Managed Objects. At the time the request is issued, it may not be known which (or how many) Managed Objects will match the scoping and filtering criteria. The ability to set the scope and filter requests is protocol dependent. For example, CMIS allows[9] for a scope of an operation to be set as part of the containment tree. Within such an operation, a filter can be placed, and only objects that match the filter will execute the operation.

We have not addressed the issue of filtering in this paper. We expect that filtering will be deemed to be a necessary functionality in the future. One solution is to add a filter argument to each RPC call. This is analogous to the instance parameter that is described below for supporting management agents. Such a parameter can be added while maintaining upward compatibility with the mapping as proposed in this paper. This is done using the version information available with DCE RPC.

Support for scoping needs to be provided by the binding mechanism. It is not directly related to the mapping between GDMO and DCE. It is rather related to the object model being used and thus outside the scope of this paper. When scoping is deemed to be necessary, this issue needs to be addressed.

4.3. The Anatomy of a GDMO RPC Definition

All of the RPC definitions that result from translating GDMO to DCE have the following form:

```
void <mo-name>_<op-name> (
        [in] handle_t handle,
        [in] char *instance,
        /* . . . */
        [out, ptr] common_err_t *c_errs );
```

Namely:

(a) All RPCs have a `void` return; output values are passed in the parameter list.

(b) The name (<*mo-name*>_<*op-name*>) of the RPC is not significant since it is not transmitted on the wire.

(c) The `handle` parameter is the standard DCE RPC binding handle.

(d) The `instance` parameter is a pointer to the relative distinguished name of the instance on which the operation is to be performed. This name should be relative to the management agent (to which the application is bound), cell relative or fully distinguished. A null value directs the operation at the bound object itself.

(e) The `c_errs` parameter is a pointer to a data structure that can contain the common errors defined in CMIS. A null pointer indicates that no common error occurred. The data structure is of the following form:

9. Both scoping and filtering are optional functional units within CMIS. An implementation can be compliant with the standard and not provide either functionality. AOM profiles specify implementation and limit the complexity of expressions.

```
typedef union switch (unsigned long localValue) {
        case 1:
                ObjectInstance noSuchObjectInstance;
        case 7:
                Attribute invalidAttributeValue;
        case 10:
                ProcessingFailure processingFailure;
        /* .
           .
           . */
} common_err_t;
```

This definition is derived from the CMIS `ERROR` definitions in [9596-1]. The original definition for `case 1` above is:

```
noSuchObjectInstance ERROR
        PARAMETER ObjectInstance
        ::= localValue 1
```

In the DCE data type, the value of the discriminator is the `localValue` assigned to the error (e.g., `1`) and the data type holding the parameter is derived from the ASN.1 type (e.g., `ObjectInstance`) using the rules described above. The field name is the same as the `ERROR` name (e.g., `noSuchObjectInstance`).

4.4. The GDMO Templates

The following sections define the mapping from each of the GDMO template types to DCE RPC definitions. In all cases it the ASN.1 data types are translated to DCE data types as described above.

4.4.1. `PARAMETER`

The `PARAMETER` template is only referenced to augment another template definition, such as an `ATTRIBUTE`, `ACTION` or `NOTIFICATION`. As such, the mapping of the `PARAMETER` template will be dependent on the place where it is referenced. The common aspect of a parameter is that it has a syntax specified in ASN.1 to carry data. In all cases the DCE mapping of this ASN.1 data type will be used to hold the value of the parameter. Examples of the mapping of the `PARAMETER` template in each context will be given in the following sections.

4.4.2. `ATTRIBUTE`

An `ATTRIBUTE` template will map to RPC definitions that allow the operations of `GET` and/or `REPLACE` to be performed on single valued attributes as well as `ADD` and/or `REMOVE` on set valued attributes. The access permitted to the attribute is specified when it is referenced in a package. Suppose there is an attribute called `server-name` that is referenced as `GET-REPLACE` in a package contained in a managed object class called `gsi`:

```
server-name ATTRIBUTE
        WITH ATTRIBUTE SYNTAX AModule.ServerName;
        BEHAVIOUR server-nameDesc;
        REGISTERED AS {1 2 3 4 5 6};
```

This maps to the following two RPC definitions:

```
void gsi_get_server_name (
        [in] handle_t handle,
        [in] char *instance,
        [out] ServerName *server_name,
        [out, ptr] common_err_t *c_errs );
```

```
void gsi_rep_server_name (
        [in] handle_t handle,
        [in] char *instance,
        [in] ServerName server_name,
        [out, ptr] common_err_t *c_errs );
```

Note that `server_name` is an `[out]` parameter in the `get` call and is passed by reference. Whereas in the `replace` call it is an `[in]` parameter, so is passed by value.

If the attribute is `DERIVED FROM` another, then it inherits its RPC definition with the addition of any new error parameters that are specified. See Section 4.4.4 ‘‘`ACTION`’’ for an example of how error parameters are added.

Any `MATCHES FOR` clause does not affect the definition of the RPC.

4.4.3. ATTRIBUTE GROUP

An `ATTRIBUTE GROUP` is mapped as a data structure containing all the attributes and a single call to get the group. For example, the following definition:

```
server-info ATTRIBUTE GROUP
        GROUP ELEMENTS
                server-name,
                server-type;
        REGISTERED AS {1 2 3 4 5 8};
```

maps to:

```
typedef struct {
        ServerName server_name;
        ServerType server_type;
} server_info;

void gsi_get_server_info (
        [in] handle_t handle,
        [in] char *instance,
        [out] server_info *server_info_group,
        [out, ptr] common_err_t *c_errs );
```

The order of the attributes in the structure is the same as the order in the original GDMO template.

Note that in the `PACKAGE` template it is possible to add more attributes to a group if it is not `FIXED`. These additional attributes will be added to the end of the same data structure in the order specified in the package. For example, consider this extract from a package:

```
ATTRIBUTE GROUPS
        server-info another-attr one-more;
```

where `another-attr` and `one-more` have syntax `AnotherAtrr` and `OneMore` respectively. This results in the following data structure for the group:

```
typedef struct {
        ServerName server_name;
        ServerType server_type;
        AnotherAtrr another_attr;
        OneMore one_more;
} server_info;
```

4.4.4. ACTION

The `ACTION` template is mapped to a single RPC call with parameters for the `PARAMETERS`, `INFORMATION SYNTAX` and `REPLY SYNTAX`. Consider a simple case that contains parameters of type `SPECIFIC-ERROR` only:

```
suspend ACTION
        PARAMETERS          not-supported, not-running;
        WITH INFORMATION SYNTAX     A-Module.Suspend;
        WITH REPLY SYNTAX   A-Module.ServerState;
        REGISTERED AS       {1 2 3 4 5 9};

not-supported PARAMETER
        CONTEXT SPECIFIC-ERROR;
        WITH SYNTAX         A-Module.ReasonCode;
        REGISTERED AS       {1 2 3 4 5 10};

not-running PARAMETER
        CONTEXT SPECIFIC-ERROR;
        WITH SYNTAX         A-Module.ServerState;
        REGISTERED AS       {1 2 3 4 5 11};
```

This maps to the following RPC and supporting `typedefs`:

```
typedef union switch (unsigned long err_code) data {
        case 1:
                not_supported not_supported_data;
        case 2:
                not_running not_running_data;
} suspend_errors_t;

void gsi_suspend (
        [in] handle_t handle,
        [in] char *instance,
        [in] Suspend info_syntax,
        [out] ServerState *reply_syntax,
        [out, ptr] suspend_errors_t *errors,
        [out, ptr] common_err_t *c_errs );
```

Note that the error codes in the union switch are numbered in the order that the parameters are specified, beginning at 1. If there are no specific errors a null pointer is passed as the value of the errors parameter.[10]

It is also possible for actions to contain `ACTION-INFO` or `ACTION-REPLY` parameters. In this case these are represented as extra parameters to the RPC call. They are grouped such that all the information parameters are together and all the reply parameters are together. Within these groups they are in the order in which they appear on the `PARAMETERS` line, with the original information and reply syntaxes coming first. For example if the `ACTION` definition were:

10. This mapping allows at most one specific error to be returned from an `ACTION`. This is consistent with CMIP which also allows only one specific error to be returned. If this restriction is not acceptable, this mapping needs to be modified.

```
suspend ACTION
        PARAMETERS      not-supported,
                        not-running,
                        time-delay,
                        active-count;
        WITH INFORMATION SYNTAX A-Module.Suspend;
        WITH REPLY SYNTAX       A-Module.ServerState;
        REGISTERED AS           {1 2 3 4 5 9};

time-delay PARAMETER
        CONTEXT ACTION-INFO;
        WITH SYNTAX             A-Module.Num-secs;
        REGISTERED AS           {1 2 3 4 5 12};

active-count PARAMETER
        CONTEXT ACTION-REPLY;
        WITH SYNTAX             A-Module.Int32;
        REGISTERED AS           {1 2 3 4 5 13};
```

the RPC definition would look like this:

```
void gsi_suspend (
        [in] handle_t handle,
        [in] char *instance,
        [in] Suspend info_syntax,
        [in] Num_secs time_delay,
        [out] ServerState *reply_syntax,
        [out] Int32 *active_count,
        [out, ptr] suspend_errors_t *errors,
        [out, ptr] common_err_t *c_errs );
```

It is possible that, when the action is referenced in a package, yet more parameters may be added. New `ACTION-INFO` and `ACTION-REPLY` parameters added this way are represented in the RPC call just as if they had been referenced in the action template itself, at the end of the information and reply parameter groups respectively. If the new parameters are `SPECIFIC-ERRORs` then their definitions are added to the end of the `union switch` data structure that is defined for the errors referenced in the original action template.

4.4.5. NOTIFICATION

The `NOTIFICATION` template must be mapped to an interface that uses whatever asynchronous notification service is available. The nature of the interface will be dependent very much on this service. Since there is no notification service provided as part of DCE and the interface to DME's NeL has not been released, this document does not propose a mapping. A mapping to the notification service should include customized routines that allow the notification provider to send a notification and the client to receive the notification. Data marshalling should be provided for both the server and client to allow them to deal only with C data structures mapped from the ASN.1 definitions in the `NOTIFICATION` template.

4.4.6. PACKAGE

The main purpose of a `PACKAGE` is to pull together a set of other GDMO templates into a unit that can be supported by a Managed Object. A number of packages are combined to form a `MANAGED OBJECT CLASS`. A package also has the unhealthy ability to add extra parameters to the referenced attributes and actions and add attributes to the referenced attribute groups (if they are not `FIXED`). The impact of these additions has already been discussed in each of the relevant template sections above.

4.4.7. MANAGED OBJECT CLASS

The `MANAGED OBJECT CLASS` template is where a number of packages are brought together to form a single unit that may be supported in its entirety by a Managed Object or its agent. This is represented in DCE as an interface that contains all of the functions to access the attributes and perform actions contained in the managed object class. The managed object class definitions are hierarchical and each class is mapped to a separate interface — an agent wishing to support a particular class will export all of the DCE interfaces in the hierarchy. For example if `MANAGED OBJECT CLASS x` is derived from `y` which is in turn derived from `top` then a DCE based management agent will export interfaces for `x`, `y` and `top`. This has the advantage that any application that is compiled to support only `top` may bind to the agent and perform operations supported by `top`.

There is an issue regarding the UUID that is used in the interface. Although each managed object class has its own object identifier, there is no static one-to-one mapping from object identifiers to UUIDs. For new GDMO definitions that are written with DCE in mind it would be possible to select object identifiers that have a decimal version of the UUID embedded in them. This is not useful, however, for existing definitions.

An example of a `MANAGED OBJECT CLASS` mapped to a DCE interface is given below:

```
gsi MANAGED OBJECT CLASS
        DERIVED FROM top;
        CHARACTERIZED BY
                gsiPackage PACKAGE
                BEHAVIOUR gsiDesc;
                ATTRIBUTES
                        server-name      GET-REPLACE,
                        server-type      GET;
                ACTIONS
                        suspend,
                        resume;
                REGISTERED AS {1 2 3 4 5 15};
        ;
REGISTERED AS {1 2 3 4 5 14};
```

It is represented by the following interface:

```
[uuid(434174b6-ebc7-11c9-9131-080009016361)]
interface Gsi_gsi {
/* gsi MANAGED OBJECT CLASS definitions */
import "Gsi__main.idl"; /* This file contains all the type
                           definitions from the ASN.1 Module */

/* Definitions from inline PACKAGE gsiPackage */
void gsi_get_server_name (
        [in] handle_t handle,
        [in] char *instance,
        [out] ServerName *server_name,
        [out, ptr] common_err_t *c_errs );

void gsi_rep_server_name (
        [in] handle_t handle,
        [in] char *instance,
        [in] ServerName server_name,
        [out, ptr] common_err_t *c_errs );

void gsi_get_server_type (
        [in] handle_t handle,
        [in] char *instance,
        [out] ServerType *server_type,
        [out, ptr] common_err_t *c_errs );
```

```
void gsi_suspend (
        [in] handle_t handle,
        [in] char *instance,
        [in] Suspend info_syntax,
        [in] Num-secs time_delay,
        [out] ServerState *reply_syntax,
        [out] Int32 *active_count,
        [out, ptr] suspend_errors_t *errors,
        [out, ptr] common_err_t *c_errs );

void gsi_resume (
        [in] handle_t handle,
        [in] char *instance,
        [out] ServerState *reply_syntax,
        [out] resume_errors_t *errors
        [out, ptr] common_err_t *c_errs );
}
```

If the package had been referenced rather than defined in-line, the interface would look exactly the same. If more than one package is specified then the definitions for each package will appear in the order that the packages appear in the managed object class.

If `CONDITIONAL PACKAGE`s are specified then all the definitions from them are also included in the interface. If a management agent wishes not to support one of the conditional packages at runtime, it can return the common error indicating that the package is not supported. In an implementation, this could be achieved by using a table of function pointers for the operations that have been mapped from conditional packages. Dummy routines could be supplied to return an error if the package is not supported, or a pointer to a routine to handle the operation could be inserted when a package is supported.

4.4.8. `NAME BINDING`

The `NAME BINDING` template has not been addressed by this proposal, since it is concerned with naming and instance creation/deletion. These aspects relate to the object model being used and, although this is an issue that needs to be addressed, it is outside the current scope of this paper.

4.4.9. `BEHAVIOUR`

The `BEHAVIOUR` template is intended for containing human readable comments and semantic restrictions. These have no bearing on the syntactic aspects of the interface. It is suggested that they be incorporated as comments at appropriate places within the DCE IDL source files.

5. PROXY

In some situations it may be necessary to manage DCE services using the CMIP protocols. This might be the case if a particular installation wishes to manage their environment using CMIS-based applications. Since the management interface to the DCE service is defined using GDMO, this becomes fairly straightforward. A proxy that translates from CMIP to DCE RPC can be built.

The proxy can be one of two types — a proxy that has knowledge of specific object classes (per-object proxy) or a generic proxy. Though we have not built a prototype, we are convinced that a per-object proxy could be automatically built from the GDMO templates. Compiler technology similar to our GDMO to RPC compiler could generate the necessary code. It may even be possible to place this code into dynamically-linked libraries for a proxy process to use.

A generic proxy would have to gain its knowledge about the objects from the metadata and create translations at runtime. For this to work, support for interpretive stubs would need to be added to DCE.

The proposal in this paper neither prescribes nor precludes either of these possible methods of creating a proxy.

6. SUMMARY

In this paper we have presented an efficient way for DME and other management systems to manage DCE-based services. By creating a mapping from GDMO to RPC we combine the benefits of a common way of defining managed objects and an efficient way of implementing the interface. The mapping allows for either direct communication with the DCE service or through a proxy. This is accomplished using DCE RPC in its intended fashion.[11]

The mapping from ASN.1 to DCE IDL is not a perfect mapping since it maps an abstract definition language to a concrete definition language. We do not feel that this will cause a problem since the limitations imposed by DCE IDL are similar to those imposed by the programming languages used to implement the services.

This paper has raised several issues. First, in the mapping proposed we do not support filtering. Support for filtering can be added without major changes to the mapping.[12] However, it would incur some overhead even when no filter was specified. The mapping for the filter information needs to be developed.

Second, there are three areas that need further specification. These are related to the object model and binding mechanism rather than to the mapping of GDMO to RPC. Though this places them beyond the scope of this paper, they are issues that should be addressed. They are:

(a) A mapping to a notification service needs to be defined. We assume that when the interface specification to DME's NeL is released, the `NOTIFICATION` template can be mapped to it.

(b) We have not defined support for a scoping (or multi-cast) service.

(c) We have not addressed naming issues in this paper. These will be addressed in a future paper.

REFERENCES

[9596-1] ISO 9596-1, ''Information Technology — Open Systems Interconnection — Common Management Information Protocol — Part 1: Specification'', June 1991.

[AppDevGd] OSF DCE Release 1.0, ''DCE Application Development Guide'', December 1991.

[KT1] N. Kincl and D. Thompson, ''Interface Modeling and Implementation within DME'', IAG-92-001, Hewlett-Packard, February 12, 1992.

[X.208] CCITT Recommendation X.208, ''Specification of abstract syntax notation one (ASN.1)'', Fascicle VIII.4, 1988.

[X.209] CCITT Recommendation X.209, ''Specification of basic encoding rules for abstract syntax notation one (ASN.1)'', Fascicle VIII.4, 1988.

AUTHORS' ADDRESSES

Norman Kincl
Hewlett-Packard Company
19410 Homestead Road, M/S 43UF
Cupertino, CA 95014-9810
USA

Internet email: kincl@nsa.hp.com
Telephone: +1-408-447-2431

11. Some proposals map the CMIP protocol onto RPC, using RPC instead of ROS. This essentially turns RPC into a transport protocol.
12. This can be done in the future, maintaining upward compatibility through DCE RPC version information.

Dean Thompson
Hewlett-Packard Company
3403 East Harmony Road, M/S 102
Fort Collins, CO 80525-9599
USA
Internet email: d_thompson@fc.hp.com
Telephone: +1-303-229-3131

Richard Webber
Hewlett-Packard Company
19410 Homestead Road, M/S 43UF
Cupertino, CA 95014-9810
USA
Internet email: webber@nsa.hp.com
Telephone: +1-408-447-2509