OSF DCE SIG
Request For Comments: 58.0

R. Viveney (DEC)
T. Hinxman (DEC)
J. Harrow (DEC)
R. Annicchiarico (DEC)
February 1994

EXTENSIONS TO DCE IDL FOR SUPPORT OF OBJECT-ORIENTATION

1. INTRODUCTION

The notion of distributed object-oriented programming is currently an active topic in the research and development communities. For example, the Object Management Group's (OMG) Common Object Request Broker Architecture and Specification (CORBA) describes a mechanism by which objects make requests and receive responses (see [CORBA]). CORBA does not specify interoperability. However, various vendors are currently implementing multiple versions of CORBA, and unless some coordination is exerted, interoperability will not result.

OSF DCE, on the other hand, fully specifies interoperability (as well an many other things not included in CORBA), and lacks only a fully object-oriented patina. This RFC outlines enhancements to the DCE IDL compiler for the purpose of providing object-oriented functionality. The features proposed here are sufficient to support CORBA on top of DCE, though this RFC stops short of indicating exactly how that should be done.

Topics covered in this paper include:

(a) Compilation Units.

(b) Parameterized Exceptions.

(c) Dynamic Invocation.

(d) `ANY` Data Type.

2. COMPILATION UNITS

Currently, the DCE IDL compiler supports only a single interface definition per input file. Since objects will, by nature, usually have several interfaces, it would be convenient to collect all these interfaces in a single file, thereby facilitating the reuse of `typedef`'s and constants. Currently, this can be achieved with the use of an `import` statement.

Hence, the DCE IDL compiler should be extended to allow several interface definitions to exist in the same input file, thus redefining a compilation unit to be either a file or an interface definition.

Furthermore, the DCE IDL compiler should be extended to allow types, constants, and imports to be declared outside the scope of an interface. All identifiers declared in this way will become available to all the interface compilation units in the file immediately succeeding the declaration.

In the following example, allowing identifiers outside the scope of an interface definition permits both interface `Put` and interface `Get` to use the `stringRef` type definition.

```
const unsigned long   maxStringLen = 10;
typedef [string] char stringRef[maxStringLen];

/* ... */
```

```
interface Put {
    void putString(
        [in] handle_t  h,
        [in] stringRef s
    );
}

interface Get {
    void getString(
        [in] handle_t   h,
        [out] stringRef s
    );
}
```

2.1. Global Name Space

All identifiers declared in an interface definition file will have a global scope relative to that file. In other words, if an interface declares a type within its definition, that type will be available to all subsequent interface definitions within that file without qualification.

In the following example, interface `Get` can use the `stringRef` type since it was previously defined and has a global scope. Note that it would be more natural to place identifier definitions outside the body of an interface definition since it enhances readability and emphasizes the global scope of all names.

```
/* ... */

interface Put {
    const unsigned long   maxStringLen = 10;
    typedef [string] char stringRef[maxStringLen];

    void putString(
        [in] handle_t  h,
        [in] stringRef s
    );
}

interface Get {
    void getString(
        [in] handle_t   h,
        [out] stringRef s
    );
}
```

3. PARAMETERIZED EXCEPTIONS

DCE IDL currently supports an exception mechanism to allow for effective handling of error conditions. A server that confronts an abnormal situation can raise an exception condition, which is then transmitted back to the client. The client can catch exception conditions and act appropriately. The current DCE IDL does not support the passing of information along with exception conditions.

The DCE IDL should be extended to allow parameters to be associated with exception conditions. The IDL syntax should be changed to allow a parameter list in the exception declaration, which appears as part of an interface attribute. The definitions of all types used in a parameter list must precede the exception declaration.

In the following example, `my_exc1` is a user-defined exception with no parameters (as supported by IDL in DCE V1.0.3) while `my_exc2` is a parameterized exception that passes a `long` and a `double`.

```
exceptions (my_exc1,
    my_exc2 (long *l, double *d)
);
```

3.1. Generated Functions

To facilitate the use of exceptions, two exception-specific functions will be generated by the IDL compiler to raise and read an exception with a parameter list. These functions will be named *exception_name*_raise() and *exception_name*_read(). Both functions will be generated with a parameter list consistent with the exception definition. The server code will make use of the *exception_name*_raise() function in order raise an exception. The client code, upon catching *exception_name*, will make use of the *exception_name*_read() function in order to read an exception parameter list into local storage. To keep the call signatures consistent between the two functions, and since the client needs to specify storage locations for the parameters, the parameters in the list must be passed by reference.

In the following example of a server application, a parity_error exception is raised while reading a file and the file name is transmitted back to the client application, along with exception notification.

```
exceptions (
    parity_error ([string] char fName[100])
);

/* server application to read a tape */
void
readTape()
{
    char *fileName;

    /* exception condition detected by server code reading
       fileName */
    parity_error_raise(fileName); /* call generated function */
}

/* idl generated exception routine */
void
parity_error_raise(char fName[100])
{
    /* form a packet containing the exception parameters */
    /* attach the packet to a per-thread context
       (use pickling) */
    RAISE(parity_error);
}

/* server stub code */
/* ... */

    /* Catch parity_error */
    /* get packet from per-thread context */
    /* call rpc_call_transmit_fault */

/* ... */
```

The client code for catching the parity_error exception would appear similar to the following:

```
/* client stub code */
/* ... */

    /* when parity_exception is received, attach it
       to per-thread storage */
    /* if no fault_status attribute specified, call
       RAISE(parity_error) */

/* ... */

/* client application to read a tape */
void
processTape()
{
    TRY {
        readTape();
    }
    CATCH (parity_error) {
        char fName[100];

        parity_error_read(fName); /* call by reference */
        printf("Caught parity error reading %s\n", fName);
        exit(1);
    }
    ENDTRY;
}

void
parity_error_read(char fName[100])
{
        /* copy exception parameter and release storage */
        rpc_ss_release_exception_params();
}
```

3.2. Releasing Parameter Storage

Note the use of the routine `rpc_ss_release_exception_params()`. This new routine will be added to the RPC run time library for the purpose of freeing storage used by the parameter data. This routine will be called by the *`exception_name`*`_read()` function automatically. If the client does not invoke the *`exception_name`*`_read()` function, the `rpc_ss_release_exception_params()` function should still be called to free storage. Once this routine is called, the exception parameters are no longer available to the client. Similarly, a second exception overrides a previous exception, and frees any storage allocated to it.

3.3. Multiple Interfaces and Common Exceptions

It may be the case that an object has several interfaces defined for it. These interfaces may have operations that want to handle the same exceptions. It would cause an error to generate the same function more than once in separate stubs that are to be linked together. To alleviate this problem, the ACF attribute, `extern_exceptions`, should be used to suppress generation of these functions in all but one of the stubs.

3.4. Fault Status Attribute

If the user chooses to use the `fault_status` ACF attribute, it would be convenient to provide a mechanism whereby the values of the parameters of a parameterized exception can be retrieved. This is particularly true if RPC is to be used with languages other than C.

A new ACF parameter attribute, `fault_id`, will be provided. It will be similar to `fault_status` in that it designates a new parameter to be added to the signature of the client stub. The type of this parameter will be `unsigned long`. If a V1.0.3-style user exception, or a parameterized user exception, is signaled by the `fault_status` parameter, the value of the `fault_id` parameter will be the ordinal number of the exception as transmitted in the fault packet.

If the `fault_status` parameter indicates a parameterized exception was received, the user can use the value of the `fault_id` parameter to determine which routine to call to get the values of the exceptions parameters and release the storage in which they are held.

3.5. NDR Implications

The NDR for a fault packet resulting from a parameterized user exception will be similar to the NDR for a fault packet from a regular user exception except that the parameter data will follow the existing fault code and exception identifier. The exception identifier is generated in the normal way. It reflects simply the position in a particular interface's exceptions list. Hence, the RPC runtime must support the transmission and reception of arbitrarily long fault packets.

3.6. RPC Runtime Issue

Currently, the datagram runtime will not accept fault data that is more than one packet long. Hence, either the datagram runtime will have to be changed, or there will be a limit on the size of the exception data.

4. DYNAMIC INVOCATION

Clients of objects may not be aware of all the various interfaces or operations an object supports. It would be unreasonable to require that all clients be recompiled whenever new interfaces are created for a particular object. Therefore, clients need some way to dynamically invoke an operation contained in an interface without prior knowledge of the interface. Clients should not have to be linked with an interface stub to use it.

To support dynamic invocation, the following routines should be added to the IDL run time library.

4.1. idl_dynamic_invoke()

```
idl_dynamic_invoke(
    rpc_binding_handle_t binding_handle,
    idl_byte             *interface_description,
    idl_ulong_int        operation_number,
    idl_void_p_t         *parameter_addresses,
    error_status_t       *p_comm_status,
    error_status_t       *p_fault_status,
    idl_ulong_int        *p_fault_id
);
```

Here, the parameters are defined as follows:

(a) `binding_handle` is the binding handle to be used for the call. Even if the interface definition specifies a binding method other than explicit handle, the call will use this parameter as the binding handle for the call.

(b) `interface_description` is a data structure describing the interface. See below for a description of how it might be obtained.

(c) `operation_number` is the operation number required by the RPC Protocol. For the first operation in the interface, this would be 0.

(d) `parameter_addresses` is a vector of addresses for the parameters of the operation. Even if the interface definition specifies that a parameter is passed by value, for dynamic invocation a parameter address must be supplied. Element `[0]` of this vector is the address to which the function result of the operation is to be delivered. For a `void` operation, element `[0]` is not used and may be `NULL`.

(e) `p_comm_status` is used for communication errors. If this parameter is `NULL`, communication errors will be reported to the client application code as exceptions. Otherwise, this parameter is the address to which the communication error status for the call is returned.

(f) `p_fault_status` is used for exceptions. If this parameter is `NULL`, exceptions at the server will be reported to the client application code as exceptions. Otherwise, this parameter is the address to which the server fault status for the call is returned. `p_fault_id` must not be `NULL` if `p_fault_status` is not `NULL` and the operation can return a user-defined exception. The exception identifier for a user-defined exception will be returned to this address.

4.2. Interface Description

An *interface description* is an opaque, network-transmissable array of bytes that describes an IDL interface. IDL will provide a mechanism for obtaining an interface definition from a stub linked into an application.

4.3. Dynamic Invocation Restrictions

A dynamic call will not be possible if the operation to be called has any of the following properties:

(a) It uses customized handle binding.

(b) It uses constructs that require the user to supply routines with names derived from data types. `[represent_as]`, `[transmit_as]` and OSF International Character support are not supported.

(c) Other ACF features have no effect on the processing of an RPC made via the `idl_dynamic_invoke()` function.

4.4. Operation Parameters

Some applications may want to dynamically determine the parameters required for an operation. IDL requires data structures whose IDL definitions are:

```
/* Handle for use by application code */
typedef idl_ulong_int idl_type_t;

#define idl_c_ulong_int        1
#define idl_c_long_int         2
#define idl_c_float            3
#define idl_c_eol              4
#define idl_c_fixed_struct     5

/* ... */

typedef struct idl_constr_type_t {
    unsigned long        size;
    [size_is(size)] byte type_description[];
} idl_constr_type_t;

typedef struct idl_dyn_param_desc_t {
    boolean           is_in;          /* TRUE if parameter is
                                         [in] */
    boolean           is_out;         /* TRUE if parameter is
                                         [out] */
    idl_type_t        param_type;     /* Indicates type of
                                         parameter */
    idl_constr_type_t *p_constr_type; /* If param has
                                         constructed type,
                                         pointer to data
                                         structure described
                                         for "any" */
} idl_dyn_param_desc_t;
```

```
typedef struct idl_dyn_op_desc_t {
    unsigned long num_params;    /* Number of parameter
                                    descriptions.  A function
                                    result description is
                                    included even for "void"
                                    operations */
    [size_is(num_params)] idl_dyn_param_desc_t
                 param_descs[]; /* Parameter descriptions */
} idl_dyn_op_desc_t;
```

and a routine of the form:

```
idl_dyn_get_op_desc(
    idl_byte          *interface_description,
    idl_ulong_int     operation_number,
    idl_dyn_op_desc_t **p_p_op_desc,
    error_status_t    *p_status
);
```

where:

(a) `interface_description` is a data structure describing the interface. How to obtain this data will be explained later.

(b) `operation_number` is the operation number required by RPC Protocol. For the first operation in the interface, this would be 0.

(c) `p_p_op_desc` is the address to which a pointer to the operation description will be returned. It is the user's responsibility to release this data structure when it is no longer required.

(d) `p_status` is the address to which the error status for the call is returned.

4.4.1. Freeing the operation description

To free the operation description, use:

```
idl_dyn_op_desc_free(
    idl_dyn_op_desc_t *p_op_desc
);
```

4.5. Structure Analysis

If a parameter is identified as having a structure type, it is possible to do some additional dynamic analysis of the parameter. To determine the structure size, use:

```
idl_dyn_get_struct_size(
    idl_constr_type_t *p_struct_type, /* Description of the
                                         structure */
    idl_ulong_int     *p_size         /* Addr where struct size
                                         is placed */
);
```

This can only be done for a structure that is not conformant.

To determine whether the structure is semantically equivalent to one that is already defined, use:

```
idl_dyn_match_struct_type(
    idl_constr_type_t *p_struct_type,  /* Description of the
                                          structure */
    idl_constr_type_t **match_list,    /* List of structure
                                          definitions that
                                          might match
                                          *p_struct_type */
    idl_ulong_int     match_list_size, /* Number of elements in
                                          match_list */
    idl_long_int      *match_index     /* Addr which receives
                                          the index in
                                          match_list of the
                                          definition which
                                          matches, or -1 if no
                                          match found */
);
```

4.6. Structures As Collections Of Fields

A structure which is a collection of scalar fields or fixed arrays can be built or read using the following machinery. The `idl_bound_pair_t` is used to hold the bounds information or data limit information for one dimension.

```
typedef struct idl_bound_pair_t {
    idl_long_int lower;
    idl_long_int upper;
} idl_bound_pair_t

idl_dyn_get_struct_handle(
    idl_constr_type_t       *p_struct_type, /* Description of
                                               the structure */
    idl_void_p_t            struct_addr, /* Address of the
                                            structure */
    idl_dyn_struct_handle_t *p_struct_handle
);
```

This routine creates a handle that can be used to build or read a structure. It is the caller's responsibility to make sure that `struct_addr` satisfies any alignment constraints for the structure.

To process the structure, use:

```
idl_dyn_next_struct_field_type(
    idl_dyn_struct_handle_t struct_handle,
    idl_type_t              *p_type
);
```

This routine returns the type of the first field of the structure that has not yet been acted upon. When all the fields have been written or read, an `idl_c_eol`, from the list of constants defined for an `idl_type_t`, is returned.

If the field is a fixed array, then a description of that array can be obtained by calling the following routine:

```
idl_dyn_array_desc_get(
    idl_dyn_struct_handle_t struct_handle,
    idl_array_desc_t        **p_p_array_desc
);
```

where `idl_array_desc_t` has the IDL description:

```
typedef struct {
    idl_constr_type_t *p_base_type;   /* Pointer to description
                                         of array base type */
    idl_ulong_int     dimensionality; /* Number of dimensions
                                         of array */
    [size_is(dimensionality)] idl_bound_pair_t
                      bounds[];       /* Bounds of the array */
} idl_array_desc_t;
```

An array description uses dynamically allocated storage, and can be released by a call to:

```
idl_dyn_array_desc_free(
    idl_array_desc_t *p_array_desc
);
```

The function:

```
idl_dyn_write_struct_field(
    idl_dyn_struct_handle_t struct_handle,
    idl_void_p_t            p_value
);
```

places the value pointed at by `p_value` into the current structure field, while the function:

```
idl_dyn_read_struct_field(
    idl_dyn_struct_handle_t struct_handle,
    idl_void_p_t            p_value
);
```

places the value of the current structure field into the location indicate by `p_value`. Finally, the function:

```
idl_dyn_destroy_struct_handle(
    idl_dyn_struct_handle_t struct_handle
);
```

is called when all the fields of the structure have been processed. It releases all resources owned by the handle.

4.7. Implementation Notes

4.7.1. Interface description file

This file will be generated by executing a script which contains the following steps.

(a) Invoke the IDL compiler with a flag indicating ‘‘interface description’’ to generate C source code for the interface description.

(b) Compile and link a C program which `#include`'s the text generated by step (a).

(c) Execute the program from step (b) to write the file.

The mechanism used for storing the data needed by this process is an issue that warrants further discussion.

5. THE ‘ANY’ DATA TYPES

An `any` data type is used for a self-describing set of data which is determined at runtime. Along with a type definition, there will be sets of routines to build and explode `any` data.

A type `idl_any_t` will be defined in `nbase.idl`. From the user's point of view this is an opaque handle which he can use to build or explode an `any` data item.

5.1. Building ‘ANY’ Data

An `any` data item is built by making a set of calls to builder functions which will be supplied as part of this extension. The routines listed here are suggestive, rather than definitive. Final definition of such routines is dependent on having a substantial amount of implementation design available. The function:

```
idl_any_build_create(idl_any_t *p_any)
```

will start the building of an `any` data item. The following routines are used to add data to the `any` data type.

```
idl_any_build_add_scalar(
    idl_any_t    any,          /* "any" data item currently
                                  being built */
    idl_type_t   scalar_type,  /* Type of scalar to be added to
                                  "any" data item */
    idl_void_p_t p_value       /* Address of data to be added */
);
```

adds a scalar data item to the `any` data item being constructed. A set of `idl_build_add_`*`scalar`*`_value` routines/macros could be layered on this routine. This routine is also used when an enumeration data item is to be added to an `any` data item. In this case `scalar_type` specifies `enumeration`.

```
idl_any_build_add_constructed(
    idl_any_t          any,              /* "any" data item
                                            currently being built
                                            */
    idl_constr_type_t *p_constr_type,    /* Type descriptor for
                                            constructed data type
                                            */
    idl_void_p_t       p_value           /* Address of data to be
                                            added */
);
```

is used to add a structure, union or pointer to an `any` data item. Adding a pointer to an `any` data item implies adding its referent to that item. The meaning of `[ptr]` is restricted in an `any` data item. Aliasing is detected only between the referents of `[ptr]` pointers within the same `any` data item.

The function:

```
idl_any_build_add_array(
    idl_any_t          any,              /* "any" data item
                                            currently being built
                                            */
    idl_constr_type_t *p_constr_type,    /* Type descriptor for
                                            the array */
    idl_void_p_t       p_value,          /* Address of array to be
                                            added */
    idl_bound_pair_t   *bounds,          /* Array bounds */
    idl_bound_pair_t   *data_limits      /* Array data limits */
);
```

will add an array to an `any` data item. If the array is not conformant, `bounds` may be `NULL`. If the array is not varying, `data_limits` may be `NULL`. Otherwise these pointers must point to as many elements as the array has dimensions. Only those fields which correspond to items that would normally be specified by attributes will be used in adding the array to the `any` data item.

Finally, the routine:

```
idl_any_build_close(idl_any_t any)
```

is called to indicate that no further data will be added to an `any` data item. It must be called before the `any` data item is transmitted.

5.2. idl_constr_type_t

For each named type in the IDL, a data item of type `idl_constr_type_t` will be written into the stub, if an appropriate IDL compiler option is specified. The name of the data item will be of the form *`typename`*`_type`.

It will be possible for application code to obtain such data from a stub and store it away for use when a definition of this type is subsequently required. The mechanism for storing and retrieving this data is outside the scope of this proposal.

The definition of `idl_constr_type_t` expressed in IDL terms is:

```
typedef struct idl_constr_type_t {
    unsigned long        size;
    [size_is(size)] byte type_description[];
} idl_constr_type_t;
```

5.3. Exploding 'ANY' Data

The recipient of an `any` data item needs a set of routines to extract data from it. It will extract the data one component at a time. The routines listed here are suggestive, rather than definitive. Final definition of such routines is dependent on having a substantial amount of implementation design available.

To get the type of the next element data:

```
idl_any_expl_get_type(
    idl_any_t  any,     /* "any" data item currently being
                           exploded */
    idl_type_t *p_type /* Receives type of next component of
                           "any" data item */
);
```

returns the type of the next component of the `any` data item. An `idl_c_eol` marker will be returned if the `any` data item has no more components.

If the type returned by `idl_any_expl_get_type` is a scalar or enumeration, it is simple to deliver the value to the user using the following routine.

```
idl_any_expl_get_scalar(
    idl_any_t    any,     /* "any" data item currently being
                             exploded */
    idl_void_p_t p_value /* Receives data value */
);
```

If the type returned by `idl_any_expl_get_type` is a constructed type, two cases are possible. Consider first the case where the receiver knows the internal structure of the constructed type. If the type is a conformant array, use:

```
idl_any_expl_get_array_size(
    idl_any_t     any,      /* "any" data item currently being
                               exploded */
    idl_ulong_int widths[]
);
```

The user must allocate storage for the array before it can be received. `widths` must be a vector with as many elements as the array has dimensions. Each element of the vector receives the value of `upper_bound - lower_bound + 1` for the corresponding dimension.

If the user knows that the component is a structure, union, array or pointer for which it has a definition, then:

```
idl_any_expl_get_constructed(
    idl_any_t    any,     /* "any" data item currently being
                             exploded */
    idl_void_p_t p_value /* Receives data value */
);
```

can be used to retrieve the component into the address pointed to by `p_value`. If the receiver does not know the internal structure of the constructed type, it may simply be discard.

```
idl_any_expl_discard_constructed(
    idl_any_t any /* "any" data item currently being exploded
                     */
);
```

If the constructed type is a structure, the receiver may want to treat it as a collection of components.

```
idl_any_expl_struct_get_type(
    idl_any_t   any,   /* "any" data item the structure belongs
                          to */
    idl_type_t *p_type /* Receives type of next component of
                          "any" data item */
);
```

returns the type of the next structure field. An `idl_c_eol` marker will be returned if the structure has no more fields. If the type returned by `idl_any_expl_struct_get_type` is a scalar or enumeration, it is simple to deliver the value to the user with:

```
idl_any_expl_struct_get_scalar(
    idl_any_t    any,    /* "any" data item the structure
                            belongs to */
    idl_void_p_t p_value /* Receives data value */
);
```

The receiver may wish to skip to the end of the structure it has been extracting fields from. This can be accomplished with:

```
idl_any_expl_skip_to_struct_end(
    idl_any_t any /* "any" data item the structure belongs to
                     */
);
```

If the type returned by `idl_any_expl_struct_get_type` is a union, pointer or array, the receiver has no knowledge of its internal structure and will need to discard the field with:

```
idl_any_expl_skip_struct_field(
    idl_any_t any /* "any" data item the structure belongs to
                     */
);
```

The other possible type that can be returned by `idl_any_expl_struct_get_type` is *begin nested structure*. In this case, calls to `idl_*_struct_*` routines will be interpreted as referring to the nested structure until the end of the nested structure has been reached. The assumption is made here that the possibility of exploding a constructed data type, other than a structure, whose internal structure is not known is not worth considering.

5.4. Other 'ANY' Management Routines

```
idl_any_destroy(idl_any_t *p_any)
```

This routine is designed for use on the client side. It releases all the memory used for the `any` data item.

5.5. Restrictions

The following IDL constructs cannot appear within an `any` data item:

(a) `[transmit_as]`.

(b) Context handles.

(c) Pipes.

No ACF feature has any effect on the construction of an `any` data item.

These restrictions are based on the following principles:

(a) Call-outs to user routines are not possible.

(b) IDL allows certain constructs only as parameters.

If arrayified pointers, that is pointers with array attributes on them, are supported, a special API for building and exploding them will be required. Similarly for non-encapsulated unions.

5.6. Identified 'ANY' Data

This is an optimization which avoids repeated transmission of the type description part of an `any` data item. After the building of an `idl_any_t` has been completed by an `idl_any_build_close()`, a call may be made to:

```
idl_any_get_type_desc(
    idl_any_t         any,
    idl_constr_type_t **p_p_type_desc
);
```

which yields a pointer to the description of the `any` type. This description may be stored in a repository or passed around the network. The functionality used to do this are outside the scope of this proposal.

A new type, `idl_identified_any_t`, is defined. The wire format for this type is:

```
typedef struct IDL_wire_identified_any_t {
    uuid_t                         type_uuid;
    idl_ulong_int                  data_size;
    [size_is(data_size)] idl_byte *data;
} IDL_wire_identified_any_t;
```

A data item of type `idl_identified_any_t` is built by first building an item of type `idl_any_t` and then converting it by calling:

```
idl_any_to_identified(
    idl_any_t            any,              /* [in] */
    uuid_t *             p_uuid,           /* [in] The UUID the
                                              sender knows
                                              represents the
                                              type description
                                              */
    idl_identified_any_t *p_identified_any /* [out] */
);
```

When a data item of type `idl_identified_any_t` is received, it must be converted to a data item of type `idl_any_t` before it can be exploded. This is done in three stages. The receiver first calls:

```
idl_any_identified_uuid(
    idl_identified_any_t identified_any, /* [in] */
    uuid_t               *p_uuid         /* [out] UUID from
                                            wire format */
);
```

Using functionality outside the scope of this specification, the receiver obtains the type description indicated by the UUID. It then calls:

```
idl_any_from_identified(
    idl_identified_any_t identified_any, /* [in] */
    idl_constr_type_t    *p_type_desc,   /* [in] */
    idl_any_t            *p_any          /* [out] */
);
```

to obtain an `any` data item that can be exploded.

5.7. Implementation Notes

5.7.1. Offset vector generation

A significant work item in the implementation of this proposal will be the generation of offset vectors from type vectors. This code has to be written on a per-platform basis.

5.7.2. New type definitions

```
/* Wire format for an idl_any_t */
typedef struct IDL_wire_any_t {
    idl_ulong_int                 type_size;
    [size_is(type_size)] idl_byte *type_vector;
    idl_ulong_int                 data_size;
    [size_is(data_size)] idl_byte *data;
} IDL_wire_any_t;

/* Local implementation of an idl_any_t */
typedef struct IDL_local_any_t {
    IDL_wire_any_t wire_rep;
    idl_void_p_t   workspace;  /* Anything needed to build
                                  "any" data item */
} IDL_local_any_t;
```

The `type_vector` field will be like a type vector fragment, with some changes. No offset vector exists.

5.7.3. Routine implementations

```
idl_any_build_create(idl_any_t *any)
```

creates the initial type vector and data buffers. As we don't know how big they need to be, chaining of working buffers will be needed.

REFERENCES

[RFC 48.0] R. Annicchiarico, DCE-RFC 48.0, ‘‘C++ Support in DCE RPC — Functional Overview’’, August 1993.

[CORBA] ‘‘The Common Object Request Broker: Architecture and Specification’’, OMG, December 1991.

AUTHORS' ADDRESSES

Robert Viveney
Digital Equipment Corporation
110 Spit Brook Road ZK2-3/Q18
Nashua, NH 03062-2698
USA
Internet email: viv@zko.dec.com
Telephone: +1-603-881-0362

Tony Hinxman
Digital Equipment Corporation
110 Spit Brook Road ZK2-3/Q18
Nashua, NH 03062-2698
USA
Internet email: hinxman@clt.enet.dec.com
Telephone: +1-603-881-2195

Jerry Harrow
Digital Equipment Corporation
110 Spit Brook Road ZK2-3/Q18
Nashua, NH 03062-2698
USA
Internet email: harrow@clt.enet.dec.com
Telephone: +1-603-881-2193

Rico Annicchiarico
Digital Equipment Corporation
110 Spit Brook Road ZK2-3/Q18
Nashua, NH 03062-2698
USA

Internet email: rico@clt.enet.dec.com
Telephone: +1-603-881-2685