DCE IDL EXCEPTION ENHANCEMENTS — A COMPARISON OF TWO MODELS

1. INTRODUCTION

This RFC is a motivation for adding an exception model to the DCE IDL that can handle arbitrary exception data. The need for an exception model is outlined and a comparison between two different approaches, the parameterized exception (PE) and the typed exception (TE) model, is made. Also a short look is taken at other existing exception models, like in CORBA, DCE thread package, C++ or the ISO 11578 draft proposed. Then the PE and the TE models are explained in greater detail and eventually their advantages and disadvantages are discussed.

1.1. The Need For A User-Defined Exception Model

Routines can fail for different reasons. Since communication and RPC specific failures are covered by the current DCE, user-defined exceptions are not provided (or only as a `long int` value in DCE 1.1). But several reasons exist for sending back a more detailed description about the kind of failure (e.g., the context, failure in another subsystem); especially the building of a CORBA ORB on top of DCE needs an exception model that can send data back to the caller, but other APIs will profit, too.

1.2. Semantics Of Exceptions In RPCs

The raising of an exception terminates the called RPC. When an exception is raised, the contents of the parameters and the return value of the RPC are undefined (except the `exc_id` parameter if defined in the ACF file).

If a user-defined exception is received by the client stub, but a termination of the RPC in that state (e.g., pipe not empty) would cause the raising of another exception by the stub, the user-defined exception has higher priority and is raised to the client.

1.3. Existing Exception Models

1.3.1. DCE 1.1[1]

The extension to the IDL compiler allows specification of a set of user-defined exceptions that may be generated by the server implementation of the interface. If an exception occurs during the execution of the server, it terminates the operation and the exception is propagated from server to client.

The goal of this approach is to provide user-defined exceptions at the interface level. Exceptions are declared via the exceptions interface attribute. With these enhancements it is possible to declare user-defined exceptions, but they are still restricted to an `unsigned long int`.

1.3.2. CORBA

The exception declaration in CORBA is similar to the declaration of structures. An exception does not need members. Routines, which can raise user-defined exceptions, must list the exception identifiers in their raise expression. There is a set of standard exceptions predefined by the ORB, which must not be listed in a raise expression.

1.3.3. C++

In the C++ exception model, an exception is an object which can be thrown and caught. ‘‘An exception is caught by specifying its type. However, what is thrown is not a type but an object. If we need to transmit extra

1. Includes user-defined exceptions without data, see [RFC 2.1]

information from the throw point to the handler we can do so by putting data into that object'' [Strous]. Because C++ is an object-oriented language, exceptions can be grouped in classes and exceptions can be derived.

1.3.4. Interface Definition Notation

The *Interface Definition Notation (IDN)* [ISO] specifies termination declarations as exceptions. The termination declaration is a parameterized declaration in the scope of an interface.

2. SHORT INTRODUCTION TO THE TWO DIFFERENT APPROACHES

Both the parameterized exception and the typed exception model are capable of associating arbitrary data with an exception.

2.1. Parameterized Exception (PE) Model

The PE model borrows the termination concept described in the ISO RPC draft. It also fits the `[out]` parameter model in the current IDL. All types and type attributes that are possible for `[out]` parameters are legal for the exception parameter, except for the `pipe` type.

2.2. Typed Exception (TE) Model

The TE model is modelled on the existing exception model in C++ and the proposal in the CORBA specification. While function parameters in DCE IDL are more general than type definitions, the TE model has some restrictions (e.g., conformant types, context handles). But therefore this approach fits better with the need of an object-oriented exception model, where exceptions are handled as objects. The view of exceptions as types is more general. It is also simpler to map to target languages other than C++ with an object or typed exception concept.

3. THE PARAMETERIZED EXCEPTION (PE) MODEL

The PE model being discussed is based on the proposal made by Vivebey, Hinxman, Harrow and Annicchiarico in [RFC 58.0].

3.1. The IDL File

3.1.1. Grammar

The model introduces the new keyword `exception`. Numbers refer to the numbers used in the description in ``OSF DCE 1.0.2: Application Environment Specification''. Only the productions which will be changed and the new productions are listed. The new productions have the prefix `N`.

```
(9)   <export> ::= <type_declarator> ;
                 | <const_declarator> ;
                 | <tagged_declarator> ;
                 | <exception_declarator> ;

(N1) <exception_declarator> ::=
         exception <Identifier> <parameter_declarators>
```

Restrictions:

(a) All the restrictions that are described for `[out]` parameter declarations in the [DCE AES] hold for the exception declarations, too.

Notes:

(a) All parameters must be passed by reference.

(b) Since an exception passes only information from the server to the client, directional attributes must not be declared.

3.1.2. Example (part of the IDL file)

```
interface foo
{
...
exception read_error( long *track, long *sector );
exception write_error( long *track, long *sector );
exception mystery_error( void );

void copy_file(...)
...
}
```

3.2. ACF File

The model introduces the new interface attribute `without_using_exceptions`. `Without_using_exceptions` causes the IDL compiler not to include the package `exc_handling.h` and not to generate the `EXCEPTION` declaration.

The model introduces a new operation attribute: `exc_id`. The `exc_id` is handled like the `fault_status`. Parameters qualified with `exc_id`, `comm_status` or `fault_status` must be `[out]` parameters in the IDL file. If the corresponding parameters do not exist in the IDL file, they will be created in the order they appear in the ACF file.

Example:

```
[without_using_exceptions] interface foo {
...
   copy_file ( [exc_id] user_status );
...
}
```

Notes:

(a) If the interface attribute `without_using_exceptions` is used, the attribute `exc_id` must be used in each operation, which can raise user-defined exceptions

(b) The attribute `exc_id` can be used without the use of `without_using_exceptions`.

3.3. The Generated Header File

A function for raising an exception is generated for each exception having the same parameter declarations as the corresponding exception.

An `EXCEPTION` declaration is also generated and the CMA exception handling package is included (unless the use of the CMA package is disabled by the ACF file). A unique (in this interface) integer value is produced for each exception and defined as an identifier. The `EXCEPTION` declaration uses the exception name as defined in the IDL file; the identifier is defined with a trailing `_id`. The identifier values are generated by incrementing the value of its predecessor by one and starting at a given offset.

A macro `RPC_RAISE` is inserted in the header file, which should be used to raise an exception rather than the generated functions. This macro hides the different functions from the user.

```
#include <exc_handling.h>
...
void copy_file(...);

/* used only by the manager (server): */
#define RPC_RAISE( exc_name )raise_ ## exc_name
/* Example:
   RPC_RAISE( read_error(&t) ) --> raise_read_error (&t) */

/* used only by the manager (after RPC_RAISE substitution): */
extern void raise_read_error( long *track, long *sector );
extern void raise_write_error( long *track, long *sector );
extern void raise_mystery_error();

/* only used by the client (and the stubs): */
EXCEPTION read_error, write_error, mystery_error;
#define read_error_id       1
#define write_error_id      2
#define mystery_error_id    3

extern int get_read_error( long *track, long *sector );
extern int get_write_error( long *track, long *sector );
...
```

Notes:

(a) The manager code should only contain calls of the `RPC_RAISE` macro and should not call the generated functions explicitly.

(b) Because of the `EXCEPTION` declaration the threads exception package has to be included (if not disabled by the ACF file).

(c) The `EXCEPTION` declaration and the identifier values are used only by the client (and by the stubs).

(d) The usage of the CMA exceptions and the examination of the identifier values are mutually exclusive.

3.4. The Manager Code

For raising a user-defined exception, the user declares variables of appropriate types. Then the variables are assigned values. Eventually, the exception can be raised by using the macro `RPC_RAISE()`.

```
void copy_file(...)
{
  ...
  /* compute ... */
  if (read_error_occurred) {
     long t,s;
     t = 47;
     s = 11;
     RPC_RAISE( read_error(&t, &s) );
  }
  if (write_error_occurred) {
     long t,s;
     t = 47;
     s = 11;
     RPC_RAISE( write_error(&t, &s) );
  }
  if (something_else_went_wrong) {
     RPC_RAISE( mystery_error() );
  }
  ...
}
```

3.5. The Generated Server Stub

The exception functions are in the generated server stub. A call to one of these functions (i.e., the manager raises an exception) passes the parameters and gives the control to the stub. Unlike normal functions, these functions will never return; they terminate by raising a CMA exception (or using a `longjump` directly), which is caught by the routine that forms the fault packet (in the same stub).

The memory management (clean-up) is the same as described in the [DCE AES] for `[out]` parameters. The only difference between `[out]` parameters and exceptions is that an exception is coded into the `Fault PDU`. The status field of the `Fault PDU` is set to the exception identifier.

> *NOTE: It is worth pointing out here that this implies a major change in the RPC protocol as it is now required to handle fragmented* `Fault PDU`*s.*

3.6. The Generated Client Stub

The client stub receives the `Fault PDU`, examines its status field and raises the associated CMA exception or passes the status code, if forced by the ACF file.

If the CMA package is used, the client stub initializes all possible exceptions once before doing the first RPC call.

A function `get_`*`exception_name`* for reading the additional data is generated and placed in the client stub. This function has as parameter a pointer to the appropriate type (pass by reference) and passes the unmarshalled data.

3.7. The Client Code

3.7.1. The client code (using the CMA exception package)

The client (user written) knows all possible exceptions, since they are defined in the IDL file. After catching an exception, the client can get the additional data by calling the corresponding function `get_`*`exception_name`*.

```
#include <exc_handling.h>
...
TRY
  copy_file(...);
CATCH ( read_error ) {
  long t, s;
  get_read_error( &t, &s );
  /* do something */
}
CATCH ( write_error ) {
  /* I'm not interested in seeing that data... */
  /* do something else */
CATCH ( mystery_error ) {
  /* If I knew what to do... */
}
CATCH_ALL() {
  /* all other possible exceptions... */
}
...
```

3.7.2. The client code (without using the CMA exception package)

The client (user written) knows all possible exceptions, since they are defined in the IDL file. The use of the CMA package is disabled by the ACF file, but an additional parameter, which holds the status code, is generated. The client can get the additional data by calling the corresponding function `get_`*`exception_name`*.

```
#include <exc_handling.h>
...
error_status_t user_status;
copy_file(..., &user_status);
switch (user_status) {
  case error_status_ok: break; /* no exception */
  case read_error_id: {
    long t, s;
    get_read_error( &t, &s );
    /* do something */
  }
  case write_error_id: {
    /* I'm not interested in seeing that data... */
    /* do something else */
  case mystery_error_id: {
    /* If I knew what to do... */
  }
  default: {
    /* all other possible (user-defined) exceptions... */
  }
}
...
```

4. THE TYPED EXCEPTION (TE) MODEL

4.1. The IDL File

4.1.1. Grammar

New keywords will be used: `exception`, `raises`. Numbers refer to the numbers used in the description in ‘‘OSF DCE 1.0.2: Application Environment Specification’’. Only the productions, which will be changed and the new productions are listed. The new productions have the prefix `N`.

```
(9)   <export> ::= <type_declarator> ;
                 | <const_declarator> ;
                 | <tagged_declarator> ;
                 | <exception_declarator> ;
(71) <op_declarator> ::= [ <operation_attributes> ]
        <simple_type_spec> <Identifier> <parameter_declarators>
        [ raises ( <Identifier_list> ) ]

(N1) <exception_declarator> ::=
        exception [ <type_attribute_list> ] <type_spec>
            <declarators>
(N2) <Identifier_list> ::= <Identifier> [ , <Identifier> ... ]
```

Restrictions:

(a) All the restrictions that are described for the type declarations in the [DCE AES] hold for the exception declarations, too.

(a) If you use the `raises` option to indicate which exceptions can occur, these exceptions must be declared.

(b) Use the `raises` option for user-defined exceptions only.

4.1.2. Example (part of the IDL file)

```
interface foo
{
...
typedef struct {
        long track;
        long sector;
        } IOerr_t;

exception IOerr_t read_error;
exception IOerr_t write_error;
exception void mystery_error;

void copy_file(...)
     raises( read_error, write_error, mystery_error );
...
}
```

4.2. ACF file

The model introduces the new interface attribute `without_using_exceptions`. This attribute causes the IDL compiler not to include the package `exc_handling.h` and not to generate the `EXCEPTION` declaration.

The model introduces a new operation attribute: `exc_id`. The `exc_id` is handled like the `fault_status`. Parameters qualified with `exc_id`, `comm_status` or `fault_status` must be `[out]` parameters in the IDL file. If the corresponding parameters do not exist in the IDL file, they will be created in the order in which they appear in the ACF file.

Example:

```
[without_using_exceptions] interface foo {
...
   copy_file ( [exc_id] user_status );
...
}
```

(a) If the interface attribute `without_using_exceptions` is used, the attribute `exc_id` must be used in each operation, which can raise user-defined exceptions (i.e., which has a non-empty `raises` clause in the IDL).

(b) The attribute `exc_id` can be used without the use of `without_using_exceptions`.

4.3. The Generated Header File

The keyword `exception` is substituted by `typedef` and `_t` is added to the exception name.

An `EXCEPTION` declaration is also generated and the CMA exception handling package is included (unless the use of the CMA package is disabled by the ACF file). A unique (in this interface) integer value is produced for each exception and defined as identifier. The `EXCEPTION` declaration uses the exception name as in the IDL file defined; the identifier is defined with a trailing `_id`. The identifier values are generated by incrementing the value of its predecessor by one and starting at a given offset.

A function for raising the exception is generated for each exception, which has as parameter a pointer to the exception type (pass by reference) or no parameter if the type is `void`.

A macro `RPC_RAISE` is inserted in the header file, which should be used to raise an exception rather than the generated functions. This macro hides the different functions to the user.

```
#include <exc_handling.h>
...
typedef struct {
        long track;
        long sector;
        } IOerr_t;

void copy_file(...);

typedef IOerr_t read_error_t;
typedef IOerr_t write_error_t;

/* used only by the manager (server): */
#define RPC_RAISE( exc_name )raise_ ## exc_name
/* Example:
   RPC_RAISE( read_error(&t) ) --> raise_read_error (&t) */

/* used only by the manager (after RPC_RAISE substitution): */
extern void raise_read_error( read_error_t *parm );
extern void raise_write_error( write_error_t *parm );
extern void raise_mystery_error();
```

```
/* only used by the client (and the stubs): */
EXCEPTION read_error, write_error, mystery_error;
#define read_error_id      1
#define write_error_id     2
#define mystery_error_id   3

extern int get_read_error( read_error_t *parm );
extern int get_write_error( write_error_t *parm );
...
```

Notes:

(a) The manager code should only contain calls to the `RPC_RAISE` macro and should not call the generated functions explicitly.

(b) Because of the `EXCEPTION` declaration the threads exception package has to be included (if not disabled by the ACF file).

(c) The `EXCEPTION` declaration and the identifier values are used only by the client (and by the stubs).

(d) The usage of the CMA exceptions and the examination of the identifier values are mutually exclusive.

4.4. The Manager Code

For raising a user-defined exception, the user declares a variable of the appropriate type, which is defined in the header file. Eventually, the exception can be raised by using the macro `RPC_RAISE()`. Obviously, the variable should contain appropriate data at this point.

```
void copy_file(...)
{
  ...
  /* compute ... */
  if (read_error_occurred) {
     read_error_t re;
     re.track  = 47;
     re.sector = 11;
     RPC_RAISE( read_error(&re) );
  }
  if (write_error_occurred) {
     write_error_t we;
     we.track  = 08;
     we.sector = 15;
     RPC_RAISE( write_error(&we) );
  }
  if (something_else_went_wrong) {
     RPC_RAISE( mystery_error() );
  }
  ...
}
```

4.5. The Generated Server Stub

The exception functions are in the generated server stub. A call to one of these functions (i.e., the manager raises an exception) passes the parameters and gives the control to the stub. Unlike normal functions, these functions will never return; they terminate by raising a CMA exception (or using a `longjump` directly), which is caught by the routine that forms the fault packet (in the same stub).

The memory management (clean-up) is the same as described in the [DCE AES] for `[out]` parameters. The only difference between `[out]` parameters and exceptions is that an exception is coded into the `Fault PDU`. The `status` field of the `Fault PDU` is set to the exception identifier.

NOTE: It is worth pointing out here that this implies a major change in the RPC protocol as it is now required to handle fragmented `Fault PDU`*s.*

4.6. The Generated Client Stub

The client stub receives the `Fault PDU`, examines its `status` field and raises the associated CMA exception or passes the status code, if forced by the ACF file.

If the CMA package is used, the client stub initializes all possible exceptions once before doing the first RPC call.

A function `get_`*`exception_name`* for reading the additional data is generated and placed in the client stub. This function has as parameter a pointer to the appropriate type (pass by reference) and passes the unmarshalled data.

4.7. The Client Code

4.7.1. The client code (with using the CMA exception package)

The client (user written) knows all possible exceptions, since they are defined in the IDL file. After catching an exception, the client can get the additional data by calling the corresponding function `get_`*`exception_name`*.

```
#include <exc_handling.h>
...
TRY
  copy_file(...);
CATCH ( read_error ) {
  read_error_t re;
  get_read_error( &re );
  /* do something */
}
CATCH ( write_error ) {
  /* I'm not interested in seeing that data... */
  /* do something else */
CATCH ( mystery_error ) {
  /* If I knew what to do... */
}
CATCH_ALL() {
  /* all other possible exceptions... */
}
...
```

4.7.2. The client code (without using the CMA exception package)

The client (user written) knows all possible exceptions, since they are defined in the IDL file. The use of the CMA package is disabled by the ACF file, but an additional parameter, which holds the status code, is generated. The client can get the additional data by calling the corresponding function `get_`*`exception_name`*.

```
#include <exc_handling.h>
...
error_status_t user_status;
copy_file(..., &user_status);
switch (user_status) {
  case error_status_ok: break; /* no exception */
  case read_error_id: {
    read_error_t re;
    get_read_error( &re );
    /* do something */
  }
  case write_error_id: {
    /* I'm not interested in seeing that data... */
    /* do something else */
  case mystery_error_id: {
    /* If I knew what to do... */
  }
  default: {
    /* all other possible (user-defined) exceptions... */
  }
}
...
```

5. DISCUSSION OF THE PE AND TE MODELS

5.1. General shortcomings of both models

In both models the user must declare all needed exceptions in the IDL specification. In the PE model the signature is given by the types, order and number of the parameters (like an RPC prototype); in the typed exception model the type of the exception is given in the IDL.

The `raises` clause in the IDL file must specify the possible exceptions for each operation, but the check that only specified exceptions are raised cannot be made at compile-time, thus raising unspecified exceptions will only be detected at run-time. The PE model does not have a `raises` clause. Every operation that is defined in the interface can use any exception that is defined in the same interface. The `raises` clause can be added to and removed from both models without changing the general meaning of the models.

Both approaches lack the possibility of:

(a) raising multiple exceptions.
It is not possible to raise two *different* exceptions together (such as `write_error` and `disk_full`).

(b) raising subtypes of exceptions.
It is not possible to raise a subtype of an exception, if this subtype is not declared separately as exception.

(c) catching supertypes of exceptions.
Since IDL (and C, which IDL maps onto) is not object-oriented, both proposals do not provide any inheritance or hierarchy of classes. The main disadvantage of both models is that it is impossible to group exceptions. Thus the client must have a `CATCH()` statement for every possible exception.

(d) raising nested exceptions.
Nested exceptions are a special case of raising multiple exceptions; this will be discussed briefly in the last chapter.

An exception can only be sent from the server to the client (caller); the exception is not caught and reraised automatically in the case of nested RPCs as in the exception model of C++. The problem exists, because the CMA exception model is not compatible with the proposed models.

5.2. Backward Compatibility

In the existing DCE[2], an RPC is terminated by a normal return statement or in the case of an error. Only the error case can cause a raise of an exception to the client. The exception can be generated by the server stub (sending a `Fault PDU` to the client) and then raised by the client stub, or raised by a communication failure. So the introduction of user-defined exceptions is additional to the existing model. Because of using the same (modified) `Fault PDU` the `status` values of new (generated) exceptions must be different from all existing values.

In the current DCE, there is also another exception model, which can be used by threads (CMA exception model). The catch statement from this model is used by the RPC exceptions if the RPC exceptions are not forced to return an additional status parameter by the ACS. Unfortunately, both PE and TE models are concerned only with RPC exceptions and use the IDL to generate specific functions. There is no mechanism to attach the additional exception data to a CMA exception; the identification of the corresponding data is done by using a derived name (e.g., `get_exc(...)`, if the name of the exception is `exc`). This also means that the CMA exception package does not profit from the proposed models.

On the other hand it can be desirable to improve the existing CMA exception model, too. This implies adding data to the existing exceptions and using the same mechanism as the proposals do.

5.3. A Closer Look At Possible Exception Data

The more detailed analysis given below applies to both approaches, typed exceptions (TE) and parameterized exceptions (PE), unless otherwise noted. All restrictions that are described in the [DCE AES] for the current IDL remain for the new productions.

The TE model has a major disadvantage in handling DCE IDL context handles, because these are only allowed as parameters in the current DCE.

Another restriction is that the TE model cannot handle more than one conformant type in the exception data.

(a) The `pipe` type.
The semantics of an exception are chosen so that an exception is a termination statement; therefore, an exception type or parameter must not have a `pipe` type as member.

(b) Type attributes.

- (i) `transmit_as (` *`xmit_type`* `)`
  There is no technical reason to forbid the use of the `transmit_as` attribute.

- (ii) `handle`
  The `handle` attribute is handled as an ordinary parameter in the meaning of the exception model.

- (iii) `align (` *`integer_size`* `)`
  What is the purpose of this attribute? I did not find any references in the [DCE AES] documentation.

- (iv) *`usage_attribute`*

  - [a] `string`
    No change.

  - [b] `context_handle`
    Since context handles can only be parameters and must not be array elements, structure or union members, the TE model can only support a context handle, if it is declared as a separate exception. Thus no additional information can be packed into this exception. The PE model however, does not suffer from this problem.

- (v) *`union_type_switch_attr`*
  No change.

---

2. Any references to DCE in this paper refer to DCE V1.0.3 unless otherwise stated.

(vi) `ptr_attr`
No change.

(c) Conformant, varying and conformant varying arrays (TE model only).
Since conformant, varying and conformant varying arrays must have at least one additional parameter or structure field with a field attribute of `last_is`, `length_is`, `min_is`, `max_is` or `size_is`, those arrays must be encapsulated in a structure, if used as exception type. A structure can only hold one conformant type.

6. HOW DO THE PROPOSALS FIT INTO OTHER EXCEPTION MODELS?

6.1. C++

Both models are not object-oriented (OO) in the current version; so, let's look at how difficult it is to move the models towards OO models.

Looking at C++ as the most probable next target language for DCE you will see that C++ already has an exception concept. Here an exception is an object, which can be thrown and caught. Since an exception is caught by specifying its type, the catch statement matches all thrown objects of this or of a derived type. So, exceptions can be grouped in classes.

By changing the target language to C++ it is probable that the IDL will be changed, too. It is likely that the class concept and inheritance will be introduced to the existing IDL.

The mapping from the TE model into C++ would collapse to the mapping of the exception types to C++ types and the user would use the C++ exception model with the throw and catch statements in a natural way. No further raise or get functions would need to be generated.

With the PE model the mapping onto C++ is not as simple. Especially building an exception class hierarchy would be more complex, if at all possible.

6.2. CORBA

Both models are supersets of the CORBA exception model. In the CORBA IDL, an exception is defined as a structure. Structured types are provided by the TE model and the PE model. To emulate the CORBA model, it is necessary to define the CORBA exception structure as a new type. In the TE model this type can be identified with an exception; in the PE model an exception must be created that has only this type as parameter.

6.3. RPC Model

In [ISO] an RPC declaration can have a termination list; this termination list is a list of parameters. Since the ISO draft does not mention the mapping to a specific language, no restrictions on these parameters are made. The PE model, being restricted to the current DCE `[out]` parameters (reference passing), is close to this model.

My understanding of an RPC is slightly different. The RPC paradigm implies that an RPC should hide its remoteness as much as possible. An RPC call should appear to the client like a normal procedure call in the host language. Only the effects caused by the remoteness should be expressed. User-defined exceptions have nothing to do with remote semantics, but with the programming style. So, if a language supports exceptions, these exceptions should be supported by the RPC, too.

Although C does not support exceptions, the DCE thread package does; therefore it would be desirable to support the same kind of exceptions in the RPC model as in the thread model and vice-versa. As stated above, C++ already has an exception model and this model is closer to the TE model than to the PE model.

7. NESTED EXCEPTIONS AND DCE

Since the semantics of an exception are defined as immediate termination of the called procedure, and both models (TE and PE) can only handle single exceptions defined in the DCE IDL file, exceptions cannot dynamically be collected at run-time and eventually raised.

To include nested exceptions at the level of DCE, the semantics of an exception must be changed so that exceptions can be collected until a special statement flushes the exception buffer. That means that an exception does not terminate the call and statements following the raising of the exception are meaningful to the application. Moreover, the server (manager code) must decide, when to flush the exception buffer, i.e., it must have knowledge about later events.

REFERENCES

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

[DCE AES] ‘‘OSF DCE 1.0.2, Application Environment Specification’’, Open Software Foundation, 1993.

[ISO] ‘‘Information Processing Systems — Open System Inteconnection — Remote Procedure Call, Parts 1-4’’, ISO/IEC 11578 Committee Draft 2, Ed: Griesmer, Yellin; August, 1992.

[RFC 2.1] J. Harrow, DCE-RFC 2.1, ‘‘Proposed Enhancements For DCE 1.1 IDL’’, July 1992.

[RFC 58.0] R. Viveney, T. Hinxman, J. Harrow, R. Annicchiarico, DCE-RFC 58.0, ‘‘Extensions To DCE IDL Support Of Object-Orientation’’, February 1994.

[Strous] B. Stroustrup, ‘‘The C++ programming language’’, 2nd edition, Addison-Wesley, Reading, MA, 1991.

AUTHOR’S ADDRESS

Jakob Hummes
University of Karlsruhe
Institut fuer Telematik
Zirkel 2
Postfach 6980
76128 Karlsruhe
Germany

Internet email: hummes@telematik.informatik.uni-karlsruhe.de
Telephone: +49-721-608-3391