ETSI TS 102 219 V1.1.1 (2003-06)

Technical Specification

Methods for Testing and Specification (MTS); The IDL to TTCN-3 Mapping

Reference
DTS/MTS-00080

Keywords
IDL, testing, TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at
http://portal.etsi.org/tb/status/status.asp

If you find errors in the present document, send your comment to:
editor@etsi.org

Copyright Notification

No part may be reproduced except as authorized by written permission.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2003.
All rights reserved.

DECTTM, **PLUGTESTS**TM and **UMTS**TM are Trade Marks of ETSI registered for the benefit of its Members.
TIPHONTM and the **TIPHON logo** are Trade Marks currently being registered by ETSI for the benefit of its Members.
3GPPTM is a Trade Mark of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners.

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Introduction

Object-based technologies (such as CORBA, DCOM, DCE) and component-based technologies (such as CCM, EJB, .NET) use interface specifications to describe the structure of an object-/component-based system and its operations and capabilities to interact with the environment. These interface specifications support interoperability and reusability of objects/components.

The techniques used for interface specifications are often called Interface Definition Language (IDL), for example CORBA IDL, Microsoft IDL or DCE IDL. These languages are comparable in their abilities to define system interfaces, operations at system interfaces and system structures to various extends. They differ in details of the object/component model.

When considering the testing of object-/component-based systems with TTCN-3, one is faced with the problem of accessing the systems to be tested via the system interfaces as described in an IDL specification. In particular, for TTCN-3 based test systems a direct import of IDL specifications into the test specifications for the use of e.g. system's interface, operation and exception definitions is prevalent to any manual transformation into TTCN-3.

The present document discusses the mapping of CORBA IDL specifications into TTCN-3. This mapping rules out the principles not only for CORBA IDL, but also for other interface specification languages. The mapping can be adapted to the details of other interface specification languages.

The Interface Definition Language (IDL) (chapter 3 in [4]) is a base of the whole Common Object Request Broker Architecture (CORBA) [4] and an important point in developing distributed systems with CORBA. It allows the reuse and interoperability of objects in a system. A mapping between IDL and a programming language is defined in the CORBA standard. IDL is very similar to C++ containing pre-processor directives (include, comments, etc.), grammar as well as constant, type and operation declarations. There are no programming language features like, e.g. **if**-statements.

The core language of TTCN-3 is defined in ES 201 873-1 [1] and provides a full text-based syntax, static semantics and operational semantics as well as a definition for the use of the language with ASN.1. The IDL mapping provides a definition for the use of the core language with IDL (figure 1).

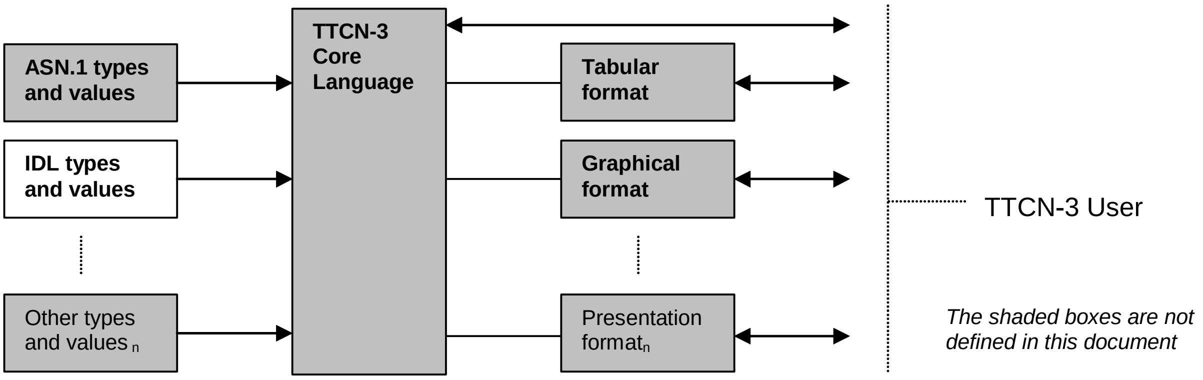

Figure 1: User's view of the core language and the various presentation formats

It makes no difference for the mapping if requested or provided interfaces are required by the test system and SUT. Hence, TTCN can be used on client and server side without modifications to the mapping rules.

The further document is structured similar to the IDL specification document to provide easy access to the mapping of each IDL element.

1 Scope

The present document defines the mapping rules for CORBA IDL (as defined in chapter 3 in [4]) to TTCN-3 (as defined in ES 201 873-1 [1]) to enable testing of CORBA-based systems. The principles of mapping CORBA IDL to TTCN-3 can be also used for the mapping of interface specification languages of other object-/component-based technologies.

The specification of other mappings is outside the scope of the present document.

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

[1] ETSI ES 201 873-1 (V2.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ISO/IEC 646: "Information technology - ISO 7-bit coded character set for information interchange".

[3] ISO/IEC 10646: "Information technology - Universal Multiple-Octet Coded Character Set (UCS)".

[4] OMG Formal Document FORMAL/2001-12-01 (2001): "The Common Object Request Broker - Architecture and Specification", Version 2.6.

[5] ITU-T Recommendation X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

3 Abbreviations

For the purpose of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
CCM	CORBA Component Model (by OMG)
CORBA	Common Object Request Broker Architecture (by OMG)
DCE	Distributed Computing Environment (by OSF)
EJB	Enterprise JavaBeans (by Sun)
IDL	Interface Definition Language
.NET	XML-based component technology (by Microsoft)
OMG	Object Management Group
OSF	Open Software Foundation
SUT	System Under Test
TTCN	Testing and Test Control Notation
XML	Extended Markup Language

4 Approach

Two different approaches can be identified: The use of either implicit or explicit mapping. The implicit mapping makes use of the import mechanism of TTCN-3, denoted by the keywords language and import. It facilitates the immediate use of data specified in other languages. Therefore, the definition of a specific data interface for each of these languages is required. Currently, ASN.1 data can be used besides the native TTCN-3 types (see clause D.1 in [1]).

The present document follows the approach of explicit mapping, i.e. IDL data are translated into appropriate TTCN-3 data. And only those TTCN-3 data are further used in the test specification.

5 Lexical Conventions

5.0 General

The lexical conventions of IDL define the comments, identifiers, keywords and literals conventions which are described below.

5.1 Comments

Comment definitions in TTCN-3 and IDL are the same and therefore, no conversion of comments is necessary.

5.2 Identifiers

IDL identifier rules define a subset of the TTCN-3 rules in which no conversion is necessary.

5.3 Keywords

When IDL is used with TTCN-3 the keywords of TTCN-3 shall not be used as identifiers in an IDL module.

5.4 Literals

The definition of literals differs slightly between IDL and TTCN-3 why some modifications have to be made. Table 1 gives the mapping for each literal type.

Table 1: Literal Mapping

Literal	IDL	TTCN
Integer	no "0" as first digit	no "0" as first digit
Octet	"0" as first digit	'FF96'O
Hex	"0X" or "0x" as first digits	'AB01D'H
Floating	1222.44E5 (Base 10)	1222.44E5 (Base 10)
Char	'A'	"A"
Wide char	L"A"	'A'
Boolean	TRUE, FALSE	true, false
String	"text"	"text"
Wide string	L"text"	"text"
Fixed point	33.33D	(see useful type IDLfixed)

IDL uses the ISO Latin-1 character set for **string** and **wide string** literals and TTCN-3 uses ISO/IEC 646 [2] for **string** literals and ISO/IEC 10646 [3] for **wide string** literals.

6 Pre-processing

Pre-processor statements are not matched to TTCN-3 because the IDL specification must be used after pre-processing it.

7 IDL specification

7.0 General

The module, interface, value and constant declaration are described now and the type and exception declaration as well as the bodies of interfaces are described later.

7.1 Module declaration

IDL modules are mapped to TTCN-3 modules. Nested IDL modules must be flattened accordingly to TTCN-3 modules.

IDL Example:

```
module identifier { body }
```

TTCN Example:

```
module identifier { body }
```

7.2 Interface declaration

Interfaces are flattened and all interface definitions are stored in one group. Import of single interface definitions from other modules via the importing group statement is possible. This can be used if inheritance is used in the IDL specification.

For each interface, a procedure-based port type is defined for the test specification. It is associated with signatures translated from attributes and operations of the interface. Since an interface can be used in operation parameters to pass object references, an **address** type is also declared in the data part. Components are used as collection of interfaces, or objects.

IDL Example:

```
interface identifier  { body }
```

TTCN Example:

```
group identifierInterface {

   ... body definitions ...

   type port identifier
        procedure { ... }

   type address identifierObject;
}
```

Interface inheritance is executed by rolling out all inherited elements. Thus, they have to be handled as defined in the interface itself. Multiple inheritance elements have to be inherited only once!

Forward references of interfaces are provided by forward referencing the according port of the interface. Local interfaces are treated as normal interfaces.

7.3 Value declaration

In contrast to type **interface**, the IDL type **value** has local operations that are not used outside the object, and are therefore not relevant from the functional testing point of view. However, since the public attributes of **value** instances are used to communicate object states, the IDL **value** type is mapped to the **record** type in TTCN-3

The example below shows how to map **valuetype** and was used from section 5.2.5 in [4].

IDL Example:

```
valuetype EmployeeRecord {
   // note this is not a CORBA::Object
   // state definition
   private string name;
   private string email;
   private string SSN;

   // initializer
   factory init(
    in string name, in string SSN );
};
```

TTCN Example:

```
type record EmployeeRecord {
   iso8859string name,
   iso8859string email,
   iso8859string SSN
}
```

7.4 Constant declaration

Constant declarations can be transformed by use of literal (see table 1) and operator mapping for floating-point and integer values (see table 2).

Table 2: Operators for constant expressions

Operator	IDL	TTCN
Unary floating-point		
Positive	+	+
Negative	-	-
Binary floating-point		
Addition	+	+
Subtraction	-	-
Multiplication	*	*
Division	/	/
Unary integer		
Positive	+	+
Negative	-	-
Bit-complement	~	not4b
Binary integer		
Addition	+	+
Subtraction	-	-
Multiplication	*	*
Division	/	/
Modulo	%	mod
Shift left	<<	<<
Shift right	>>	>>
Bitwise and	&	and4b
Bitwise or	|	or4b
Bitwise xor	^	xor4b

IDL Example:

```
const long number = 017; // 017 == 0xF == 15
const long size = ( ( number << 3 ) % 0x1F ) & 0123;
```

TTCN Example:

```
const long number := "17"O;
const long size := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;
```

8 Type declaration

8.0 General

Type declaration mapping will be shown in the following clauses.

A construct for naming data types and defining new types by using the keyword **typedef** is provided by IDL. This can be done under TTCN-3 via the keyword **type**, too.

To enhance readability and to provide a clear distinction, mapped IDL data types get the prefix IDL and the extension attribute '**variant**" as done in TTCN-3 for type **IDLfixed** (see clause E.2.4.0 in [1]).

8.1 IDL basic types

8.1.0 General

IDL basic data types are mapped to predefined or useful types in TTCN-3.

8.1.1 Integer and floating-point types

Integer and floating-point types are mapped onto the corresponding useful types **short**, **unsignedshort**, **long**, **unsignedlong**, **longlong**, **unsignedlonglong**, **IEEE754float**, **IEEE754double**, and **IEEE754extdouble**.

IDL Example:

```
const long    size       = ( ( number << 3 ) % 0x1F ) & 0123;
const float   decimal    = 15.7;
```

TTCN Example:

```
const long           size     := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;
const IEEE754float   decimal  := 15.7;
```

8.1.2 Char and wide char type

The IDL **char** and **wide char** type represent a single and wide character. They are mapped to the self defined type **iso8859char** and type **universal char**.

IDL Example:

```
const char letter      =  'ABCD';
const wchar wideLetter =  L'ABCD';
```

TTCN Example:

```
type universal char iso8859char (char ( 0,0,0,0 ) .. char ( 0,0,0,255)) with { variant "8 bit" };
const iso8859char letter      := "ABCD";
const universal char wideLetter := "ABCD";
```

8.1.3 Boolean type

The IDL **boolean** type is equivalent to the TTCN-3 **boolean** type.

IDL Example:

```
const boolean isValid = TRUE;
```

TTCN Example:

```
const boolean isValid = true;
```

8.1.4 Octet type

Octet cannot be mapped onto an integer type because it has the special feature that it will not change its internal ordering if transferred between different system architectures. To represent it **octet** is mapped to **octetstring**.

IDL Example:

```
const octet data = 0x55;
```

TTCN Example:

```
const octetstring data = '55'H
```

8.1.5 Any type

The IDL **any** type is mapped onto **anytype** in TTCN-3 which was especially introduced for this mapping.

IDL Example:

```
typedef any AllTypes;
```

TTCN Example:

```
type anytype AllTypes;
```

8.2 Constructed types

8.2.0 General

IDL provides the three constructed types **struct, union,** and **enum**. Recursive construction of types is only permitted with the **sequence** template.

8.2.1 Struct

struct is used to collect ordered data in one place where it is mapped onto **record** in TTCN-3.

IDL Example:

```
typedef struct NC {
    string id;
    string kind;
} NameComponent;
```

TTCN Example:

```
type record NameComponent {
     iso8859string id,
     iso8859string kind
}
```

8.2.2 Discriminated unions

In IDL, unions are discriminated to determine the actual type. Therefore, a **record** type is used, which contains two members. The first one stores the discriminator information using an enumeration type. The second member is a TTCN-3 **union** type which members are defined according to the specified IDL union members.

IDL Example:

```
union MyUnion switch( long ) {
    case 0 : boolean b;
    case 1 : char c;
    case 2 : octet o;
    case 3 : short s; };
```

TTCN Example:

```
type union MyUnionType  {
    boolean b,
    iso8859string c,
    octetstring o,
    short s }

type enumerated MyUnionEnumType {
    boolean_b, iso8859string_c, octetstring_o, short_s
}

type record MyUnion {
    MyUnionEnumType kind,
    MyUnionType value
}
```

8.2.3 Enumerations

Enumerations are equally defined in IDL and TTCN-3.

IDL Example:

```
enum NotFoundReason {
     missing_node,
     not_context,
     not_object };
```

TTCN Example:

```
type enumerated NotFoundReason {
    missing_node,
    not_context,
    not_object }
```

8.3 Template types

8.3.0 General

IDL supports the template types **sequence**, **string**, **wide string** and **fixed** type.

8.3.1 Sequence

IDL **sequence** is mapped to **record of** in TTCN-3 to maintain order and to allow unbounded sequences.

IDL Example:

```
typedef sequence<NameComponent> Name;
```

TTCN Example:

```
type record of NameComponent Name;
```

8.3.2 String and wstring

string and **wstring** types are sequences of **char** and **wchar**. Therefore, **string** and **wstring** are mapped to the useful type **iso8859string** and **universal charstring**.

IDL Example:

```
const string name =       "My String";
const wstring wideName =  L"My String";
```

TTCN Example:

```
const iso8859string name  := "My String";
const universal charstring wideName := "My String";
```

8.3.3 Fixed types

The **fixed** type represents a fixed-point decimal number. It is mapped to the corresponding useful type **IDLfixed** in TTCN-3 (see clause E.2.4.0 in [1]).

IDL Example:

```
typedef fixed<12,7> myFix;
```

TTCN Example

```
template IDLfixed myFixTemplate := { 12, 7, ? };  // e.g. in module definition part
var IDLfixed myFix := { 12, 7, "12345.1234567" }; // e.g. in module control part
```

8.4 Complex declarator

8.4.0 General

The last kind of type declarators are the complex **array** and **native** types.

8.4.1 Arrays

IDL **array** is equal to the TTCN-3 **array** type.

IDL Example:

```
typedef long NumberList[100];
```

TTCN Example:

```
type long NumberList[100];
```

8.4.2 Native types

Native types are used to allow implementation of dependent types. TTCN-3 provides the type **address** to address entities inside a SUT. Hence, **address** can be used for mapping of type **native** and concrete implementation is left to the user.

IDL Example:

```
typedef native MyNativeVariable;
```

TTCN Example:

```
type MyNativeVariable address;
```

9 Exception declaration

In IDL, exceptions are used in conjunction with operations to handle exceptional conditions during an operation call. Thus, a special struct-like **exception** type is provided which has to be associated with each operation that can trigger this exception. TTCN-3 also supports the use of exceptions with procedure calls by binding it to signature definitions. However, it provides no special **exception** type. Hence, exceptions are defined by using type **record**.

A definition of an **exception** is shown in the following example. The use of exception binding in signature definitions and exception catching is shown in the context of operation declaration.

IDL Example:

```
exception NotFoundException {
          NotFoundReason why;
          Name rest_of_name; };
```

TTCN Example:

```
// definition of an exception type
type record NotFoundException {
     NotFoundReason why,
     Name rest_of_name }

// definition of a template for the
// defined exception type
template NotFoundException
  NotFoundExceptionTemplate ( NotFoundReason reason, Name name ) := {
    why := reason,
    rest_of_name := name }
```

10 Operation declaration

Apart from attributes, operations are the main part of interface definitions in IDL and are used, for instance, in the CORBA scheme as procedures which can be called by clients. Procedure calls in general are supported by TTCN-3 by means of synchronous communication operations which are used in combination with ports.

IDL supports an optional **oneway** attribute for operations which implies best-effort invocation semantics without a guarantee of delivery but with a most-once invocation semantics. Message or procedure-based ports can be used for **oneway** procedures because both would be a valid mapping based upon IDL. However, the use of procedure-based ports for **oneway** procedures is recommended because the IDL specification does not guarantee that **oneway** calls are non-blocking or asynchronous. Furthermore, CORBA implements **oneway** procedures by synchronous communication, too. Use of non-blocking or blocking procedures for **oneway** operations is left to the user. Mapped **oneway** operations acquire an additional **variant** attribute (see example).

The parameter attributes **in**, **inout** and **out** describe the transmission direction of parameters and can be mapped directly to the communication parameter attributes in TTCN-3 because they have the exact same semantics.

A **raise** expression specifies all exceptions which can be thrown by an operation. It can be mapped directly to TTCN-3 because it can be indicated by the procedure signature definition by specifying an exception. Nevertheless, each operation can trigger a standard exception.

A **context** expression provides access to local properties of the called operation. These properties consist of a name and a string value. The **context** expression can be matched by redefining the operation with the context parameters included in the operation parameters (see section 4.6, [4]). The additional parameter must be of type **array** containing a type **record** for each context parameter. The **record** itself contains two variables of type **string** for the context name and value.

IDL Example:

```
// not found exception is defined in section "exception declaration"

string remoteProc1( in long Par11, out long Par12, inout string name1 )
     raises( NotFound )
     context( "MyContext1" );
```

```
// oneway procedure: no return value and no inout or out allowed!!!
oneway void remoteProc2( in long Par21, in long Par22, in string name2 );
```

TTCN Example:

```
// only operation definition

type record IDLContextElement {
    iso8859string name,
    iso8859string value_
}

type record of IDLContextElement IDLContext;

signature RemoteProcSignature1(
             in long Par11, out long Par12,
             inout charstring name1, in IDLContext context )
          return iso8859string
          exception( NotFoundException );

signature RemoteProcSignature2(
             in long Par21, in long Par22,
             in iso8859string name2 )
          with { variant "IDL:oneway FORMAL/01-12-01 v.2.6" };

type port RemoteProcPort procedure {
          out RemoteProcSignature1;
          out RemoteProcSignature2
}

type component CorbaSystem {
     port RemoteProcPort PCO
}
```

11 Attribute declaration

An **attribute** is like a set- and get-operation pair to access a value. If an attribute is marked as **readonly**, only the get-operation is used. Therefore, attribute mapping can be done by the operation mapping.

12 Names and scoping

The name definition scheme of IDL does not collide with the name definition in TTCN-3. Scoping is more restrictive in IDL than in TTCN-3, where the IDL scoping rules have to be mapped appropriately to allow seamless mapping. IDL uses nested scopes for modules, interfaces, structures, unions, operations and exceptions and identifiers are scoped in types, constants, enumeration values, exceptions, interfaces, attributes and operations. The hierarchical scopes in TTCN-3 are **module**, control part of module, **function**, **testcase** and statement blocks within control part of **module**, **function** and **testcase**.

Furthermore, TTCN-3 supports no overloading of identifiers so that no identifier name can be used more than once in a scope hierarchy. However, IDL allows redefinition of self defined types if defined inside a **module**, **interface** or **valuetype**. Hence, identifiers have to be mapped by using their path name including all **interface** and **valuetype** names as designated in IDL and TTCN-3. The use of module names is not necessary because they are reflected by the TTCN-3 module structure. An underscore is used as a separator and existing underscores are doubled.

To indicate the special treatment of TTCN-3 statements derived from IDL, TTCN-3 provides a new mechanism to attach attributes to language elements. The use of attributes makes code more readable and require no special naming scheme. Therefore, the **variant** attribute can be used to indicate the derivation of types from IDL and the special treatment for encoding by the test system. This is used in TTCN-3 for the **IDLfixed** useful type:

```
type record IDLfixed {
                            unsignedshort   digits,
                            short   scale,
                            charstring  value_
                        }
                        with { variant "IDL:fixed FORMAL/01-12-01 v.2.6" };
```

Names of new types which are specially defined for the IDL mapping and their use in conjunction with IDL shall always begin with the word IDL to provide better distinction.

Annex A:
Void

Annex B (informative): Examples

B.1 Example

B.1.0 General

The following example shows how a mapping would look like if a complete IDL and TTCN-3 specification, including a test case, is used. It is only intended to give an impression of how the different elements have to be mapped and used in TTCN-3.

Some parts are used from the CORBA standard like the Naming Service with slight modifications to cover more IDL elements.

B.1.1 IDL specification

```
module ttcnExample
{
    // ***********
    // Basic Types
    // ***********
    const long    number     = 017;  // 017 == 0xF == 15
    const long    size       = ( ( number << 3 ) % 0x1F ) & 0123;
    const float   decimal    = 15.7;

    const char    letter     =  'A';
    const wchar   wideLetter =  L'A';

    const boolean isValid    = TRUE;
    const octet   anOctet    = 0x55; // limited to 8 bit

    const string  myName     =  "my name";
    const wstring wideMyName =  L"my name";

    typedef string MyString;

    // *****************
    // Constructed Types
    // *****************
    typedef struct NC {
        MyString id;
        MyString kind;
    } NameComponent;

    union MyUnion switch( long ) {
        case 0 : boolean b;
        case 1 : char c;
        case 2 : octet o;
        case 3 : short s;
    };

    enum NotFoundReason { missing_node,
                          not_context,
                          not_object };

    // **************
    // Template Types
    // **************
    typedef sequence <NameComponent> Name;

    typedef sequence <NameComponent> Key;

    typedef fixed<12,7> Fix;
```

```
    // ******************
    // Complex Declarator
    // ******************
    typedef long NumberList[100];

    native MyNativeVariable;

    // ********************
    // Valuetype Definition
    // ********************

    valuetype StringValue string;

    valuetype EmployeeRecord {
        // note this is not a CORBA::Object
        // state definition
        private string name;
        private string email;
        private string SSN;

        // initializer
        factory init(in string name, in string SSN);
    };

    // ********************
    // Interface Definition
    // ********************
    interface NamingContext {
        attribute string object_type;
        readonly attribute Key external_form_id;

        exception NotFound {
            NotFoundReason why;
            Name rest_of_name;
        };

        MyString bind( in Name n, inout Object obj, out Object myObj )
            raises( NotFound ) context ( "Hostname" );

        oneway void rebind( in Name n, in Object obj );

    }; // end of interface NamingContext

}; // end of module ttcnExample
```

B.1.2 Derived TTCN-3 specification

```
module ttcnExample {

    // ********************************
    // Mapping of the IDL Specification
    // ********************************

    // **********************
    // Mapping of Basic Types
    // **********************
    const long number           := '17'O ;
    const long size             := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;
    const IEEE754float decimal  := 15.7;

    type universal char iso8859char (char ( 0,0,0,0 ) .. char ( 0,0,0,255))
         with { variant "8 bit" };

    const iso8859char          letter     := "A";
    const universal char       wideLetter := "A";

    const boolean              isValid    := true;
    const octetstring          anOctet    := '55'H;

    const iso8859string        myName     := "my name";
    const universal charstring wideMyName := "my name";

    type iso8859string MyString;
```

```
// *****************
// Constructed Types
// *****************

// ******
// Struct
// ******

type record NameComponent {
    MyString id,
    MyString kind
};

// *****
// Union
// *****
type union MyUnion  {
    boolean b,
    is08859char c,
    octetstring o,
    short s
};

// ***********
// Enumeration
// ***********
type enumerated NotFoundReason {
    missing_node,
    not_context,
    not_object
}

// ********
// Sequence
// ********
type record of NameComponent Name;
type record of NameComponent Key;

//******
// Fixed
// *****
// see also using of fixed in testcase below
template IDLfixed fixTemplate := { 12, 7, ? };


// ******************
// Complex Declarator
// ******************

type long numberList[100];

// see using of native in testcase below


// ********************
// Valuetype Definition
// ********************
type iso8859string StringValue;

type record EmployeeRecord {
    iso8859string name,
    iso8859string email,
    iso8859string SSN
};

// ********************
// Interface Definition
// ********************
type record IDLContextElement {
    iso8859string name,
    iso8859string value_
}

type record of IDLContextElement IDLContext;


group NamingContextInterface {
```

```
    type address NamingContextObject;

    // attribute object_type
    signature ObjectTypeGetSignature() return iso8859string;
    signature ObjectTypeSetSignature( in iso8859string object_type );

    template ObjectTypeSetSignature ObjectTypeSetSignatureTemplate := {
        object_type := "my object type"
    }

    //
    // attribute external_from_id
    //
    signature ExternalFormIdGetSignature() return Key;

    // exception notFoundException
    type record NotFoundException {
        NotFoundReason why,
        Name rest_of_name
    }

    template NotFoundException
        NotFoundExceptionTemplate ( NotFoundReason reason, Name name ) := {
            why          := reason,
            rest_of_name := name
        }

    //
    // bind procedure
    //
    signature BindSignature( in Name n, inout address obj, inout address  myObj,
                             in IDLContext context ) return MyString
        exception( NotFoundException );



    template BindSignature BindTemplate ( charstring object, IDLContext con ) := {
        n        := "name",
        obj      := object,
        myObj    := *,
        context  := con
    }


    //
    // rebind procedure
    //
    signature RebindSignature( in Name n, in address obj )
         with { variant "IDL:oneway FORMAL/01-12-01 v.2.6" };

    template RebindSignature RebindTemplate ( address object ) := {
        n   := "name",
        obj := object
    }


    type port NamingContext procedure {
        out ObjectTypeGetSignature;
        out ObjectTypeSetSignature;
        out ExternalFormIdGetSignature;
        out BindSignature;
        out RebindMessageType;
    }
}

// component is necessary for test case
type component CorbaSystemInterface {
    port NamingContext PCO;
}

// somewhere has main test component MyMTC to be defined


// *******************
// Testcase Definition
// *******************
testcase MyNamingServiceTestCase() runs on MyMTC system CorbaSystemInterface {
```

```
// examples to show how above definitions can be used inside a
// testcase definition

var CorbaSystemInterface myCorbaSystem := CorbaSystemInterface.create;
connect( self:NamingContextPCO, myCorbaSystem:PCO );
myCorbaSystem.start;

//
// Fixed Type
//
var IDLfixed fix := { 12, 7, "12345.1234567" };

//
// Native
//
var address MyNativeVariable;


//
// Procedure Calls
//
var MyString   myResult1;
var Key        myResult2;
var MyString   myResult3;
var address object, myObject, resultObject, resultMyObject;

var IDLContextElement contextElement := {
    name := "Hostname",
    value_ := "disen"
}

var IDLContext contextParameter := { contextElement };

//
// procedure get object_type
//
NamingContextPCO.call( ObjectTypeGetSignature )
{
    [] NamingContextPCO.getreply( ObjectTypeGetSignature value * )
        -> value myResult1 {}
}

//
// procedure  set object_type
//
NamingContextPCO.call( ObjectTypeSetSignatureTemplate );


//
// procedure get external_from_id
//
NamingContextPCO.call( ExternalFormIdGetSignature )
{
    [] NamingContextPCO.getreply( ExternalFormIdGetSignature value * )
        -> value MyResult2 {}
}


//
// procedure bind (with template)
//
NamingContextPCO.call( BindTemplate( object, contextParameter ) )
{
    [] NamingContextPCO.getreply( BindTemplate( * ) value * )
        -> value myResult3
        param( resultObject, resultMYObject ) sender mySender {}

    [] NamingContextPCO.catch( BindSignature,
                    NotFoundExceptionTemplate )
    {
        setverdict( fail );
        stop;
    }

}


//
```

```
        // procedure bind (without template)
        //
        NamingContextPCO.call(
                   BindSignature:{ myName, object, myObject, contextParameter } )
        {
            [] NamingContextPCO.getreply( BindSignature:{ -, *, myObject }
            value * ) -> value myResult3 param( resultObject, resultMYObject ) sender mySender {}
        }

        //
        // procedure rebind
        //
        NamingContextPCO.call( RebindSignature:{ myName, object} ); // or use a template


        //
        // raising an exception
        //

        // this would be used to raise an exception inside of procedure bind
        // if defined by TTCN-3 (if used on server side).
        var NotFoundException myNotFoundException := {
            why          := missing_node,
            rest_of_name := "noname"
        }

        NamingContextPCO.raise( BindSignature, myNotFoundException );

    } // end of testcase MyNamingServiceTestCase

} // end of module ttcnExample
```

Annex C (informative): Mapping Lists

C.1 IDL keyword and concept mapping list

Table 3 lists the mapping of keywords and concepts of IDL to TTCN-3 keywords or concepts. Literal and operator mapping can be seen in table 1 and 2.

Table 3: Conceptual list of IDL mapping

IDL	TTCN-3
`FALSE`	`false`
`Object`	`address`
`TRUE`	`true`
`abstract`	has to be rolled out
`any`	`anytype`
`array`	`array`
`attribute`	get (and set) operation
`boolean`	`boolean`
`char`	`iso8859char` `(self defined type)`
`const`	`const`
`context`	additional procedure parameter of type record
`enum`	`enumerated`
`exception`	`record`
`fixed`	`IDLfixed`
`float`	`IEEE754float`
`double`	`IEEE754double`
`long double`	`IEEE754extdouble`
`in`	`in`
`inout`	`inout`
`interface`	`group, port`
`local`	`---`
`long`	`long`
`long long`	`longlong`
`module`	`module`
`native`	`address`
`octet`	`octetstring`
`oneway`	operation with `variant` attribute
operation	`signature` for procedure
`out`	`out`
`raises`	`exception`
`readonly`	only a get-operation for the attribute
`sequence`	`record of`
`short`	short
`string`	iso8859string
`struct`	record
`typedef`	type
`union`	record, enumerated, union
`unsigned long`	unsignedlong
`unsigned long long`	unsignedlonglong
`unsigned short`	unsignedshort
`valuetype`	`record`
`wchar`	`universal char`
`wstring`	`universal charstring`

C.2 Comparison of IDL, ASN.1, TTCN-2 and TTCN-3 data types

IDL	ASN.1	TTCN-2	TTCN-3
Object	ObjectInstance (X.500 Distinguished name)	IA5String	address
any	ANY DEFINED BY [5] or SEQUENCE {typecode, anyValue}	CHOICE	anytype
array	SEQUENCE OF (with sizeConstraint subtype)	SEQUENCE SIZE(n) OF	array
boolean	BOOLEAN	BOOLEAN	boolean
char	GraphicString	GraphicString or IA5String(SIZE(1))	iso8859char (self defined type)
enum	ENUMERATED	ENUMERATED	enumerated
exception	SPECIFIC ERRORS	SEQUENCE	record
fixed	See note	See note	IDLfixed
float	REAL	See note	IEEE754float
double	REAL	See note	IEEE754double
long double	REAL	See note	IEEE754extdouble
long	INTEGER	INTEGER	long
long long	INTEGER	INTEGER	longlong
native	See note	See note	address
octet	OCTET STRING	OCTET STRING (SIZE(1))	octetstring
sequence	SEQUENCE OF (with optional sizeConstraint subtype for IDL bounds)	SEQUENCE OF	record of
short	INTEGER	INTEGER	short
string	GraphicString	GraphicString	iso8859string
struct	SEQUENCE	SEQUENCE	record
union, switch, case	CHOICE (with ASN.1 TAGS)	SEQUENCE	record, enumerated, union
unsigned long	INTEGER	INTEGER	unsignedlong
unsigned long long	INTEGER	INTEGER	unsignedlonglong
unsigned short	INTEGER	INTEGER	unsignedshort
valuetype	See note	See note	record
wchar	See note	GraphicString or BMPString(SIZE(1))	universal char
wstring	See note	GraphicString	universal charstring
NOTE: Mapping of this type was not considered.			

Annex D (informative): Bibliography

M. Ebner, A. Yin, and M. Li (2002): "Definition and Utilisation of OMG IDL to TTCN-3 Mappings". In *TESTING OF COMMUNICATING SYSTEMS XIV - Application to Internet Technologies and Services*, ed. I. Schieferdecker, H. König and A. Wolisz. IFIP, Kluwer Academic Publishers, pp. 443-458. ISBN 0-7923-7695-1

M. Ebner (2001): "A Mapping of OMG IDL to TTCN-3". SIIM Technical Report SIIM-TR-A- 01-11, Institute for Telematics, Medical University of Lübeck, Germany. Schriftenreihe der Institute für Informatik/Mathematik.

M. Ebner (2001): "Mapping CORBA IDL to TTCN-3 based on IDL to TTCN-2 mappings". In Proceedings of the 11th GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems, Bruchsal, Germany, 21.-22. June 2001. International University in Germany.
http://www.i-u.de/fbt2001/.

A. Yin (2001): "Testing Operation-Based Interfaces Exemplified for CORBA with ADL and TTCN-3". Diplomarbeit, Telecommunication Network Group, Faculty of Electrical Engineering and Computer Science, Technical University Berlin, Germany.

A.Yin, I. Schieferdecker and M. Li (2001): "Mapping of IDL to TTCN-3". Technical Report, Fraunhofer Institute for Open Communication Systems (FOKUS), Germany.

ISO/IEC 9646-3: "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

History

Document history		
V1.1.1	June 2003	Publication