Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
Part 8: The IDL to TTCN-3 Mapping
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, please send your comment to one of the following services:
http://portal.etsi.org/chaircor/ETSI_support.asp

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 2006.
All rights reserved.

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

Intellectual Property Rights...........................................................................................................5
Foreword............................................................................................................................................5
Introduction ......................................................................................................................................5

1 Scope ...........................................................................................................................................7
2 References ...................................................................................................................................7
3 Abbreviations ...............................................................................................................................7
4 Approach .....................................................................................................................................8
5 Lexical Conventions ....................................................................................................................8
  5.1 Comments ..............................................................................................................................8
  5.2 Identifiers ...............................................................................................................................8
  5.3 Keywords ...............................................................................................................................8
  5.4 Literals ...................................................................................................................................8
6 Pre-processing ...............................................................................................................................9
7 IDL specification .........................................................................................................................9
  7.1 Module declaration ................................................................................................................9
  7.2 Interface declaration ..............................................................................................................10
  7.3 Value declaration ..................................................................................................................11
  7.4 Constant declaration ..............................................................................................................11
8 Type declaration ..........................................................................................................................12
  8.1 IDL basic types .....................................................................................................................12
    8.1.1 Integer and floating-point types ......................................................................................12
    8.1.2 Char and wide char type ...............................................................................................12
    8.1.3 Boolean type ................................................................................................................12
    8.1.4 Octet type ....................................................................................................................13
    8.1.5 Any type .......................................................................................................................13
  8.2 Constructed types ..................................................................................................................13
    8.2.1 Struct ...........................................................................................................................13
    8.2.2 Discriminated unions .................................................................................................13
    8.2.3 Enumerations .............................................................................................................15
  8.3 Template types .......................................................................................................................15
    8.3.1 Sequence ....................................................................................................................15
    8.3.2 String and wstring .....................................................................................................15
    8.3.3 Fixed types ................................................................................................................16
  8.4 Complex declarator ...............................................................................................................16
    8.4.1 Arrays .........................................................................................................................16
    8.4.2 Native types ...............................................................................................................16
9 Exception declaration ..................................................................................................................16
10 Operation declaration ................................................................................................................17
11 Attribute declaration ................................................................................................................18
12 Names and scoping .....................................................................................................................18

Annex A (informative): Examples ..................................................................................................20
  A.1 Example ................................................................................................................................20
    A.1.1 IDL specification ........................................................................................................20
    A.1.2 Derived TTCN-3 specification ..................................................................................21

Annex B (informative): Mapping lists ..........................................................................................26
B.1 IDL keyword and concept mapping list ....................................................................................... 26
B.2 Comparison of IDL, ASN.1, TTCN-2 and TTCN-3 data types ............................................................. 27
Annex C (informative): Bibliography ...................................................................................................... 28
History .................................................................................................................................................. 29
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 ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS), and is now submitted for the ETSI standards Membership Approval Procedure.

The present document is part 8 of a multi-part deliverable. Full details of the entire series can be found in part 1 [1].

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 extents. 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).
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.

NOTE: While any hyperlinks included in this clause were valid at the time of publication ETSI cannot guarantee their long term validity.


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


3 Abbreviations

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

- ASN.1: Abstract Syntax Notation One
- CCM: CORBA Component Model
- CORBA: Common Object Request Broker Architecture
- DCE: Distributed Computing Environment

NOTE: By OMG.
NOTES:  
By OSF.

EJB  
Enterprise JavaBeans

NOTE:  
By Sun.

IDL  
Interface Definition Language
NET  
XML-based component technology

NOTE:  
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

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

<table>
<thead>
<tr>
<th>Literal</th>
<th>IDL</th>
<th>TTCN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>no &quot;0&quot; as first digit</td>
<td>no &quot;0&quot; as first digit</td>
</tr>
<tr>
<td>Octet</td>
<td>&quot;0&quot; as first digit</td>
<td>FF96'O</td>
</tr>
<tr>
<td>Hex</td>
<td>&quot;OX&quot; or &quot;0x&quot; as first digits</td>
<td>'AB01'D'H</td>
</tr>
<tr>
<td>Floating</td>
<td>1222.44E5 (Base 10)</td>
<td>1222.44E5 (Base 10)</td>
</tr>
<tr>
<td>Char</td>
<td>'A'</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>Wide char</td>
<td>L&quot;A&quot;</td>
<td>&quot;A&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>TRUE, FALSE</td>
<td>true, false</td>
</tr>
<tr>
<td>String</td>
<td>&quot;text&quot;</td>
<td>&quot;text&quot;</td>
</tr>
<tr>
<td>Wide string</td>
<td>L&quot;text&quot;</td>
<td>&quot;text&quot;</td>
</tr>
<tr>
<td>Fixed point</td>
<td>33.33D</td>
<td>(see useful type IDLfixed)</td>
</tr>
</tbody>
</table>


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

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.

As one IDL module can contain many nested IDL modules where several nested modules can have equal names in different scopes, these names can clash. Hence, module names identifiers are to be used which are composed of the identifiers of the upper level IDL modules (from hierarchical point of view) and the nested IDL module name, separated one from each other by two underscores.

According the IDL scoping rules nested modules have access to the scope of upper level modules. As there are no nested modules in TTCN-3, TTCN-3 modules have to import upper level modules. For avoiding name clashes, a prefix for the imported definitions composed of the identifier of the module from which it is imported shall be used. The prefix and the identifier are separated by a dot (.) as defined in TTCN-3.

IDL EXAMPLE:

```idl
module identifier1 {
  typedef long mylong1;

  module identifier2 {
    typedef string mystring2;
    typedef mylong1 mylong2;

    module identifier3 {
      typedef mylong1 long_from_module_1;
      typedef mystring2 string_from_module_2;
      typedef mylong2 long_from_module_1_2;
    }
  }
}
```
TTCN EXAMPLE:

module identifier1 {
    type long mylong1;
}

module identifier1__identifier2 {
    import from identifier1 all;
    type iso8859string mystring2;
    type identifier1.mylong1 mylong2;
}

module identifier1__identifier2__identifier3 {
    import from identifier1 all;
    import from identifier1__identifier2 all;
    type identifier1.mylong1 long_from_module_1;
    type identifier1__identifier2.mystring2 string_from_module_2;
    type identifier1__identifier2.mylong2 long_from_module_1_2;
};

7.2 Interface declaration

Interfaces are flattened and all interface definitions are stored in one group. In contrast to interfaces in IDL, groups in TTCN-3 do not create a scope. Therefore, prefixes for all identifiers of type definitions inside of the interface shall be used, which are a combination of the interface name and two underscores as the prefix.

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.

An IDL attribute is mapped to two signatures: one for the setting of a value and one for getting it. These signatures have names composed of the prefix (interface name and two underscores), attribute name and the word "Set" (except for "readonly") or "Get" correspondingly.

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 {
    attribute long attributeId;
    void operationname (in string param_value) raises (ExceptionType);
    ... other body definitions ...
};

TTCN EXAMPLE:

group identifierInterface {
    signature identifier__attributeIdGet () return long;
    signature identifier__attributeIdSet (in long identifier__attributeId);
    signature identifier__operationname (in iso8859string identifier__param_value)
      exception (ExceptionType);
    ... other 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! As normally an inherited IDL interface uses types defined in the module, usually it is essential to import the complete mapped TTCN-3 module. All inherited elements have to be rolled out directly in the TTCN-3 group for the interface, even if the inheritance is multiple.

Forward references of interfaces are provided by forward referencing the according port of the interface. Local interfaces are treated as normal interfaces. However it is recommend not to use forward references and to move a TTCN-3 definition of the interface (group) to a place where a forward definition is used first time.
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 valuertype and was used from section 5.2.5 in [4].

IDL EXAMPLE:

```idl
valuertype 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:

```ttcn
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>
<thead>
<tr>
<th>Operator</th>
<th>IDL</th>
<th>TTCN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary floating-point</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Positive</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Negative</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Binary floating-point</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td><strong>Unary integer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Positive</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Negative</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bit-complement</td>
<td>~</td>
<td>not4b</td>
</tr>
<tr>
<td><strong>Binary integer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>Modulo</td>
<td>%</td>
<td>mod</td>
</tr>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td>Bitwise and</td>
<td>&amp;</td>
<td>and4b</td>
</tr>
<tr>
<td>Bitwise or</td>
<td></td>
<td>or4b</td>
</tr>
<tr>
<td>Bitwise xor</td>
<td>^</td>
<td>xor4b</td>
</tr>
</tbody>
</table>
8 Type declaration

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

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:

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

TTCN EXAMPLE:

```ttcn
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:

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

TTCN EXAMPLE:

```ttcn
type universal char iso8859char (char ( 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:

```idl
const boolean isValid = TRUE;
```
TTCN EXAMPLE:

```tcl
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:

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

TTCN EXAMPLE:

```tcl
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:

```tcl
typedef any AllTypes;
```

TTCN EXAMPLE:

```tcl
type anytype AllTypes;
```

### 8.2 Constructed types

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:

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

TTCN EXAMPLE:

```tcl
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.

In addition, two types are defined to express the link between discriminator's type and union's type: a type to reflect the
discriminating type of a union and an enumeration to distinguish the discriminated cases. Using the information
provided by these type definitions, the marshalling/unmarshalling for discriminated unions is possible in an
unambiguous manner: to encode or decode a union value, we use the value of the kind field to resolve the
corresponding chosen option and calculate then the real value for the discriminator by resolving this value in the
discriminator enumeration.
IDL EXAMPLE 1:

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

TTCN EXAMPLE 1:

type long MyUnion__Switch;

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
}

IDL EXAMPLE 2:

Enum MyDiscr {
    BOOLEAN_DISCR,
    CHAR_DISCR,
    OCTET_DISCR,
    SEQ_DISCR,
    SHORT_DISCR
};

union MyUnion switch( MyDiscr ) {
    case BOOLEAN_DISCR : boolean b;
    case SHORT_DISCR : short s;
}

TTCN EXAMPLE 2:

type MyDiscr MyUnion__Switch;

type enumerated MyUnion__CasesType {
    case_BOOLEAN_DISCR,
    case_SHORT_DISCR
}

type union MyUnionType {
    boolean b,
    short s
}

type enumerated MyUnionEnumType {
    boolean_b,
    short_s
}

type record MyUnion {
    MyUnionEnumType kind_,
    MyUnionType value_}
8.2.3 Enumerations

Enumerations are equally defined in IDL and TTCN-3.

IDL EXAMPLE:

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

TTCN EXAMPLE:

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

8.3 Template types

IDL supports the template types `sequence`, `string`, `wstring` 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 1:

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

TTCN EXAMPLE 1:

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

IDL sequences with a specified maximum size are mapped to `record of` with limited number of elements to maintain order and restrict the maximum number of elements.

IDL EXAMPLE 2:

```idl```
typedef sequence<NameComponent, maximum_size> Name;
```

TTCN EXAMPLE 2:

```ttcn```
type record length (0, maximum_size-1) 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:

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

TTCN EXAMPLE:

```ttcn```
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

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:**

type 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.

Several new identifiers are generated during transformation of IDL types by adding to the original IDL type identifier suffixes like: "Type", "Enum", "Object", "Interface", etc. This approach and the use of TTCN-3 keywords in IDL modules can cause a name clashes, which are to be resolved by a suffix ".":

IDL EXAMPLE:

interface identifier {
    ... body definitions ...
};

//an example of the identifier, which can cause a name clash
typedef long identifierObject;

TTCN EXAMPLE:

group identifierInterface {
    ... body definitions ...
    type port identifier procedure { ... }
    //the suffix '_' is used only where necessary
    //to resolve the name clash
    type address identifierObject_;
}
type long identifierObject;
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 \texttt{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 \texttt{IDL\_fixed} useful type:

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

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 (informative):
Examples

A.1 Example

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.

A.1.1 IDL specification

```idl
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
// ***********************

typedef StringValue string;

typedef 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 module ttcnExample

A.1.2 Derived TTCN-3 specification

module ttcnExample {
  import from IDLaux all;

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

  // ****************
  // Mapping of Basic Types
  // ****************

  "const long number := oct2int('17'O);
  const long size := oct2int(int2oct(oct2int(int2oct(number,4)<<3) mod
  hex2int('1F'H),4) 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 := hex2oct('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,
    iso8859char 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 NamingContext__object_typeGet () return iso8859string;
    signature NamingContext__object_typeSet ( in iso8859string NamingContext__object_type );
    template NamingContext__object_typeSet ObjectTypeSetSignatureTemplate := {
        object_type := "my object type"
 types

 type record NamingContext__NotFoundException
  NotFoundReason why,
  Name rest_of_name
 }

template NamingContext__NotFoundException
NamingContext__NotFoundExceptionTemplate ( NotFoundReason reason, Name name ) := {
  why := reason,
  rest_of_name := name
 }

 // bind procedure

 // signature NamingContext__BindSignature( in Name n, inout address obj, inout address
myObj,
    in IDLContext context ) return MyString
exception(NamingContext__NotFoundException);

 template NamingContext__BindSignature
NamingContext__BindTemplate ( charstring object, IDLContext con ) := {
  n := { {"name", ""} },
  obj := object,
  myObj := ?,
  context := con
 }

 // rebind procedure

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

 template NamingContext__RebindSignature
NamingContext__RebindTemplate ( address object ) := {
  n := { {"name", ""} },
  obj := object
 }

type port NamingContext procedure {
  out NamingContext__object_typeGet;
  out NamingContext__object_typeSet;
  out NamingContext__external_form_idGet;
  out NamingContext__BindSignature;
 }

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

 // somewhere has main test component MyMTC to be defined
type component MyMTC {
  port NamingContext NamingContextPCO;
 }

 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,
                              NamingContext__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 NamingContext__NotFoundException myNotFoundException := {
  why := missing_node,
  rest_of_name := "noname"
}

NamingContextPCO.raise( BindSignature, myNotFoundException );

} // end of testcase MyNamingServiceTestCase
Annex B (informative):
Mapping lists

B.1 IDL keyword and concept mapping list

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

<table>
<thead>
<tr>
<th>IDL</th>
<th>TTCN-3</th>
<th>IDL</th>
<th>TTCN-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>false</td>
<td>module</td>
<td>module</td>
</tr>
<tr>
<td>Object</td>
<td>address</td>
<td>native</td>
<td>address</td>
</tr>
<tr>
<td>TRUE</td>
<td>true</td>
<td>octet</td>
<td>octetstring</td>
</tr>
<tr>
<td>abstract</td>
<td>has to be rolled out</td>
<td>oneway</td>
<td>operation with variant attribute</td>
</tr>
<tr>
<td>any</td>
<td>anytype</td>
<td>operation</td>
<td>signature for procedure</td>
</tr>
<tr>
<td>array</td>
<td>array</td>
<td>out</td>
<td>out</td>
</tr>
<tr>
<td>attribute</td>
<td>get (and set)</td>
<td>raises</td>
<td>exception</td>
</tr>
<tr>
<td></td>
<td>operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>readonly</td>
<td>only a get-operation for the attribute</td>
</tr>
<tr>
<td>char</td>
<td>iso8859char</td>
<td>sequence</td>
<td>record of</td>
</tr>
<tr>
<td></td>
<td>(self defined type)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td>const</td>
<td>short</td>
<td>short</td>
</tr>
<tr>
<td>context</td>
<td>additional procedure parameter of type record</td>
<td>string</td>
<td>iso8859string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>enum</td>
<td>enumerated</td>
<td>struct</td>
<td>record</td>
</tr>
<tr>
<td>exception</td>
<td>record</td>
<td>typedef</td>
<td>type</td>
</tr>
<tr>
<td>fixed</td>
<td>IDLfixed</td>
<td>union</td>
<td>record, enumerated, union</td>
</tr>
<tr>
<td>float</td>
<td>IEEE754float</td>
<td>unsigned long</td>
<td>unsignedlong</td>
</tr>
<tr>
<td>double</td>
<td>IEEE754double</td>
<td>unsigned long long</td>
<td>unsignedlonglong</td>
</tr>
<tr>
<td>long double</td>
<td>IEEE754extdouble</td>
<td>unsigned short</td>
<td>unsignedshort</td>
</tr>
<tr>
<td>in</td>
<td>in</td>
<td>valuetype</td>
<td>record</td>
</tr>
<tr>
<td>inout</td>
<td>inout</td>
<td>wchar</td>
<td>universal char</td>
</tr>
<tr>
<td>Interface</td>
<td>group, port</td>
<td>wstring</td>
<td>universal charstring</td>
</tr>
<tr>
<td>local</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>longlong</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### B.2 Comparison of IDL, ASN.1, TTCN-2 and TTCN-3 data types

**Table B.2**

<table>
<thead>
<tr>
<th>IDL</th>
<th>ASN.1</th>
<th>TTCN-2</th>
<th>TTCN-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>ObjectInstance (X.500</td>
<td>IA5String</td>
<td>address</td>
</tr>
<tr>
<td></td>
<td>Distinguished name)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>ANY DEFINED BY [5] or</td>
<td>CHOICE</td>
<td>anytype</td>
</tr>
<tr>
<td></td>
<td>SEQUENCE {typecode,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>anyValue}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>SEQUENCE OF (with</td>
<td>SEQUENCE SIZE(n) OF</td>
<td>array</td>
</tr>
<tr>
<td></td>
<td>sizeConstraint subtype)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>GraphicString</td>
<td>GraphicString or</td>
<td>iso8859char (self</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IA5String(SIZE(1))</td>
<td>defined type)</td>
</tr>
<tr>
<td>enum</td>
<td>ENUMERATED</td>
<td>ENUMERATED</td>
<td>enumerated</td>
</tr>
<tr>
<td>exception</td>
<td>SPECIFIC ERRORS</td>
<td>SEQUENCE</td>
<td>record</td>
</tr>
<tr>
<td>fixed</td>
<td>See note</td>
<td>See note</td>
<td>IDLfixed</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
<td>See note</td>
<td>IEEE754float</td>
</tr>
<tr>
<td>double</td>
<td>REAL</td>
<td>See note</td>
<td>IEEE754double</td>
</tr>
<tr>
<td>long double</td>
<td>REAL</td>
<td>See note</td>
<td>IEEE754extdouble</td>
</tr>
<tr>
<td>long</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>long</td>
</tr>
<tr>
<td>long long</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>longlong</td>
</tr>
<tr>
<td>native</td>
<td>See note</td>
<td>See note</td>
<td>address</td>
</tr>
<tr>
<td>octet</td>
<td>OCTET STRING</td>
<td>OCTET STRING</td>
<td>octetstring</td>
</tr>
<tr>
<td></td>
<td>(SIZE(1))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequence</td>
<td>SEQUENCE OF (with optional</td>
<td>SEQUENCE OF</td>
<td>record of</td>
</tr>
<tr>
<td></td>
<td>sizeConstraint subtype for</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IDL bounds)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>short</td>
</tr>
<tr>
<td>string</td>
<td>GraphicString</td>
<td>GraphicString</td>
<td>iso8859string</td>
</tr>
<tr>
<td>struct</td>
<td>SEQUENCE</td>
<td>SEQUENCE</td>
<td>record</td>
</tr>
<tr>
<td>union, switch, case</td>
<td>CHOICE (with ASN.1 TAGS)</td>
<td>SEQUENCE</td>
<td>record,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>enumerated, union</td>
</tr>
<tr>
<td>unsigned long</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>unsignedlong</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>unsignedlonglong</td>
</tr>
<tr>
<td>unsigned short</td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>unsignedshort</td>
</tr>
<tr>
<td>valuetype</td>
<td>See note</td>
<td>See note</td>
<td>record</td>
</tr>
<tr>
<td>wchar</td>
<td>See note</td>
<td>GraphicString or</td>
<td>universal char</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BMPString(SIZE(1))</td>
<td>charstring</td>
</tr>
<tr>
<td>wstring</td>
<td>See note</td>
<td>GraphicString</td>
<td>universal charstring</td>
</tr>
</tbody>
</table>

**Note:** Mapping of this type was not considered.
Annex C (informative):
Bibliography


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.


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

<table>
<thead>
<tr>
<th>Document history</th>
</tr>
</thead>
<tbody>
<tr>
<td>V3.2.1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>