SIIM-TR-A-01-11

SIIM Technical Report

A Mapping of OMG IDL to TTCN-3

by

Michael Ebner

Schriftenreihe der Institute für

Informatik/Mathematik

Serie A

12th July 2001

Medizinische
Universität zu Lübeck
Technisch-Naturwissenschaftliche Fakultät

Email: ebner@informatik.mu-luebeck.de

Phone: +49-451-500-3725
Fax: +49-451-500-3722

A Mapping of OMG IDL to TTCN-3

Michael Ebner

12th July 2001

Abstract

A widely used middleware for Internet based distributed systems is CORBA where interfaces are described with IDL. TTCN is used as a standardised test description language in the telecommunication area. The current version of TTCN, version 3, is among others designed to test CORBA based systems. This requires a mapping of IDL to TTCN-3 and therefore, a new mapping for TTCN-3 in contrast to the existing one for TTCN-2 gets necessary. This report provides a mapping of IDL to TTCN-3 and a comparison to the TTCN-2 mappings.

Contents

1. Introduction **1**

1.1. Interface Definition Language (IDL) 2

1.2. Tree and Tabular Combined Notation (TTCN) 4

1.3. Remainder 5

2. Lexical Conventions **5**

3. Preprocessing **6**

4. IDL Specification **7**

4.1. Module Declaration 7

4.2. Interface Declaration 7

4.3. Value Declaration 8

4.4. Constant Declaration 9

5. Type Declaration **10**

5.1. IDL Basic Types 10

5.2. Constructed Types 12

5.3. Template Types 14

5.4. Complex declarator 15

6. Exception Declaration **16**

7. Operation Declaration **16**

8. Attribute Declaration **19**

9. Names and Scoping **20**

10. Conclusion **21**

A. IDL Concept Mapping List **23**

B. Derived TTCN-3 data types **25**

C. Comparison of IDL, ASN.1, TTCN-2 and TTCN-3 data types **27**

D. Complete Example **29**

D.1. IDL 29

D.2. TTCN-3 30

References **40**

1. Introduction

Nowadays, conformance and functional testing is widely used in the telecommunication area. However, testing of distributed systems based on Internet technology is not evolved that much. This gets especially more and more important if we have a look on the increasing amount of provided services and transfered data with Internet based technology. Furthermore, traditional telecommunication services will develop to Internet based services to provide more powerful services and to create new services. Testing of this new Internet based applications is crucial for their success as it is in the telecommunication area.

A widely used middleware for Internet based distributed systems is the *Common Object Request Broker Architecture* (CORBA) which uses an object oriented concept (OMG 2001). The interfaces of CORBA *objects* are described using the *Interface and Definition Language* (IDL) (ITU-T 1997*b*, OMG 2000). In the telecommunication area the *Tree and Tabular Combined Notation* (TTCN) is used as a standardised test description language (ETSI 2001*a*). In addition, the current version of TTCN, version 3, is also designed to test CORBA based systems to satisfy the demand for testing Internet based distributed systems. Hence, TTCN-3 can be used seamless to write tests for both application areas. TTCN is, for instance, used in the CORVAL2 (CORBA Validation 2) project to test interoperability of different *Object Request Broker* (ORB) implementations (Leach 2000).

Thus, using TTCN to test CORBA based applications is a natural step in testing distributed systems. Because TTCN requires informations about the application interface it is quite natural if we could use the IDL definition for it. Hence, a mapping of IDL to TTCN to describe the used interfaces gets required. This has already been done for TTCN-2, but TTCN-3 provides now, for instance, synchronous communication and new data types. Therefore, a mapping of IDL to TTCN-3 has to be designed to make use of this new features.

Using TTCN to test CORBA based applications requires a mapping of IDL to TTCN to describe the used interfaces. This was also done formerly for TTCN-2 but TTCN-3 provides new features as stated before (Li 1998, Mednonogov 2000, Mednonogov, Kari, Martikainen and Malinen 2000, Schieferdecker, Li and Hoffmann 1998). Furthermore, it is intended to provide a direct data type mapping where IDL data types are mapped directly to TTCN data types and not to ASN.1 data types as it is used in (Mednonogov 2000, Mednonogov, Kari, Martikainen and Malinen 2000). Therefore, the old mapping of IDL to TTCN-2 has to be redesigned to make use of these new features. It was intended to describe a mapping which is independent of implementation details wherewith it can be used in general. It is assumed that a CORBA/TTCN gateway executes the concrete mapping between the TTCN test suite and the CORBA based *System Under Test* (SUT).

IDL and TTCN-3 will be now described in more detail and afterwards the remainder of this document gets explained.

1.1. Interface Definition Language (IDL)

There exist several standards which define a *Interface Definition Language* (IDL). However, the IDL specification in the *Common Object Request Broker Architecture* (CORBA) is the most popular one and others are mostly derived from it (ISO/IEC 1999, ITU-T 1997*b*, OMG 2001). Furthermore, we are interested in using it for CORBA and therefore, only CORBA IDL resp. OMG IDL is mentioned here.

CORBA is a part of the *Object Management Architecture* (OMA), the framework defined by the *Object Management Group* (OMG). OMA is a *Distributed Object Management* (DOM) architecture. It gives an overview and a base for discussion about the expected components. The heart of the OMA is the *Object Request Broker* (ORB) which is the communication infrastructure of the whole framework for the distributed environment. It allows access to distributed objects (see figure 1). The defined protocols and interfaces are common possibilities to communicate with the ORB and to use standardised services, facilities and domain. The implementation aspects are not specified because in order to be OMG conform it is necessary to implement the interfaces only. This makes it possible to choose non object-orientated programming languages and to interconnect different languages across the ORBs.

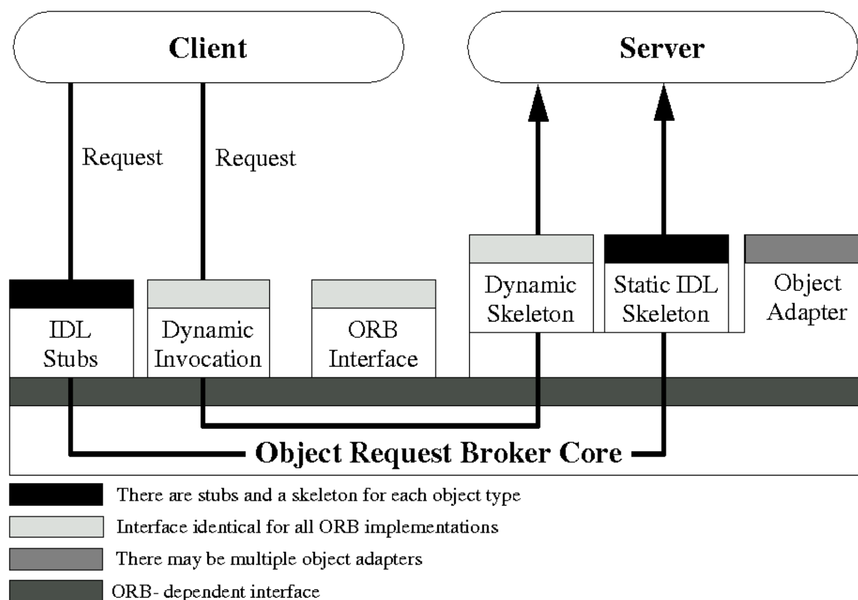

Figure 1: A CORBAobject request

The object model of CORBA is the elementary part of the whole standard. It is derived from the abstract *Core Object Model* of the OMA. It describes the view, called interface, on each object. This view is described using the IDL of the OMG for CORBA (OMG 2001, chap. 3). It is a language to *describe* interfaces in an implementation language independent manner. It shall not describe implementation characteristics like behaviour, instances or relationships. Therefore, the following constructs are defined:

Constants to assist with type declarations

Data type declarations to use for parameter typing

Attributes which allow getting and setting of a value of a particular type

Operations which take parameters and return values

Interfaces which group data type, attribute, and operation declarations

Modules for name space separation

Thus, each *client* can invoke remote operations on an object without knowing about the implementation details (see figure 1).

Therefore, IDL is a base of the whole CORBA standard 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.

Data Types IDL supports the most basic data types from C++but there are no `int` and *references* in IDL. Instead, `string`, `boolean` and `any` are available. Some data types have a different specification like `char` which is a type on its own in CORBA. The *constructed* types `enum`, `struct`, `array` and `union` are similar to C++. The *template* type `sequence` is a variable length array of elements of one, but any, IDL type. The type `string` is like `sequence` but it only supports ASCII ISO-Latin characters. `any` is like `Object` in *Java* a placeholder for *any* possible IDL type.

Modules and Interfaces The main concept behind IDL are interfaces. Each interface may contain constants, types, attributes, exceptions and operations for one object. A module is a method to separate name spaces and may contain any IDL construct. Each IDL construct is automatically *public* according to the object orientated concept. (Multiple-) Inheritance is only permitted for interfaces and not for modules. It is not forbidden to derive interfaces in a cycle.

Attributes and Operations Each client knows the IDL interface specification of each *object* containing all information about the *object*. Attributes are like variable definitions but they behave like operations in CORBA. Each *read-write* attribute gets a *set-* and *get* function and each *read-only* attribute gets a *get* function. The main part of an `interface` are the supported *operations*. An *operation* declaration consists of an operation attribute that specifies the invocation semantics, the type of the operation result, the operation name, a parameter list, optional exceptions and optional context expressions.

Exceptions are an easy programming concept to handle the distributed system's characteristics, especially errors. The context expression allows the client to transfer context specific information, like security context information for the security service.

1.2. Tree and Tabular Combined Notation (TTCN)

Tree and Tabular Combined Notation (TTCN) is the third part of the *Conformance Testing Methodology and Framework* (CTMF) standard for the specification of test suites for conformance testing. Currently, the new version of TTCN, called TTCN-3, was finally standardised to replace TTCN-2 (ETSI 2001*a*, ISO/IEC 1998*b*).

TTCN is designed for functional, black-box testing and to describe *Abstract Test Suites* (ATS) which are independent of a concrete test platform. Therefore, special interfaces between a *System Under Test* (SUT) and the ATS defined via TTCN are required to make a test suite executable. First, there has to be an *Abstract Test System Interface* (ATSI) which defines the view of the ATS. The access points between ATS and SUT are called *Point of Control and Observation* (PCO). Secondly, there is a *Real Test System Interface* required which maps the ATSI to the SUT (see figure 2).

Test configuration in TTCN is done by a *Main Test Component* (MTC), the master of the test execution. The MTC controls all other test components called *Parallel Test Component* (PTC). PTCs can be dynamically created whereas the MTC is created automatically by each test case execution. Test components in TTCN-3 communicate with each other via ports[1] which are modeled as infinite FIFO queues to store incoming calls. Communication between *Test Components* and test system is also done via ports[2] (see figure 2).

TTCN-2 was designed to test networks which are conform to the ISO *Open Systems Interconnection* (OSI) *Reference Model*. However, TTCN-3 improves concepts of TTCN-2 and introduces new concepts to be a test description language for reactive system tests over a variety of communication platforms such as CORBA based platforms. Therefore, OSI terminology and concept like *Abstract Service Primitive* (ASP) and *Packet Data Unit* (PDU) and conformance testing peculiarities have been removed as far as it is required to achieve this goal. Additionally, *constraint* handling is replaced by *templates* which provide parameterisation and matching mechanism. Another important feature in TTCN-3 is the enhanced communication concept which supports now procedure-based communication to provide synchronous communication beside the old message-based communication which is asynchronous. Through this enhancement writing tests for systems using IDL, like CORBA based systems, gets much easier. Beside these modifications a test execution control part, module and grouping concept, new data types, etc. are introduced to provide, e.g., better control and grouping mechanism. Use of data types defined via ASN.1 is also possible in TTCN-2. However, TTCN-3 has integrated some ASN.1 data types into the language itself.

As the name TTCN states, a tabular form is used in TTCN-2 for the user. However, TTCN-3 abandons the tabular form for the user and uses instead a text-based language which is comparable to an implementation language like C. This new core language is used as base for document interchange and for a tabular representation format, too (ETSI 2001*b*). A graphical representation format is also planned.

If TTCN-3 is used for testing systems with interfaces specified via IDL this interface definition can be used as ATSI. Therefore, the mapping suggestion in the next sections could be used to

[1] in TTCN-2 via *Communication Points* (CP)
[2] in TTCN-2 via PCOs

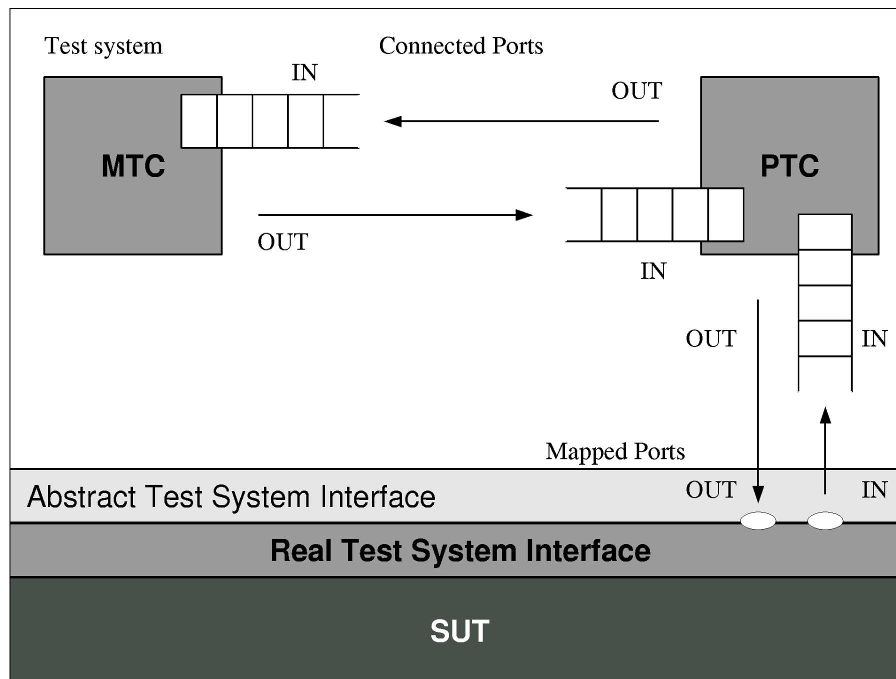

Figure 2: Conceptual view of a typical TTCN-3 testing configuration

generate ATS parts automatically. This would effect definitions like data types and signatures for procedures. Hence, interface modifications could be seamless introduced into the *static part* of TTCN test suites which would improve consistence and allows easier testing of CORBA based systems. Further information concerning TTCN-2 and TTCN-3 can be found in (J. Grabowski, Wiles, Willcock and D. Hogrefe 2000, Schieferdecker and Grabowski 2000). Especially dynamic aspects have not been considered so far because they are not that important for the mapping of IDL to TTCN.

1.3. Remainder

The further document is structured similar to the IDL specification document (OMG 2000) to provide easy access to the mapping of each IDL element. Each mapping consists of rules to map an IDL element and rules which have to be considered in the IDL definition to prevent conflicts with TTCN-3 rules. Furthermore, there can be suggestions how to use the provided features of IDL interfaces in TTCN-3, for example, the possibility of raising an exception and the exception type definition is defined in IDL but not catching of exceptions itself.

2. Lexical Conventions

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

Comments Comments can be defined in IDL and TTCN-3 by using the pair "/*" and "*/" for comment blocks or "//" for end of line comments like defined in ISO C++ (ISO/IEC 1998*c*).

Identifiers Identifiers in IDL and TTCN-3 consist of alphabetic, digit and underscore characters where the first character must be an alphabetic character and all characters are significant. However, there is no case distinction in IDL but the spelling has to be consistent throughout the whole definition. TTCN-3 uses case sensitive identifiers and therefore, the IDL rule defines a sub set of the TTCN-3 rule.

Keywords It has to be made sure that no TTCN-3 keywords (see in (ETSI 2001*a*, app. A.1.5)) are used as identifiers in the IDL definition if a seamless mapping has to be guaranteed. However, identifiers can be renamed, for example, by appending a special prefix or suffix in case of a conflict with keywords in TTCN-3.

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

Literal	IDL Convention	TTCN-3 Convention
Integer	no "0" as first digit	no "0" as first digit
Octet	"0" as first digit	'FF96'O[1]
Hex	"0X" or "0x" as first digits	'AB01D'H
Floating	1222.44E5 (Base 10)	1222.44E5 (Base 10)
Char	't'	"t"
Boolean	TRUE, FALSE	true, false
String	"text"	"text"
Wide string	"text"	"text"
Fixed point	33.33D	Not available (see section 5.3, page 15)

Table 1: Literal mapping

IDL uses the *ISO Latin-1 character set* for `string` and `wide string` literals and TTCN-3 uses ISO/IEC 646 for `string` literals and ISO/IEC 10646 for `wide string` literals (ISO/IEC 1990, 1993, 1998*a*).

3. Preprocessing

IDL preprocessing is defined by the ISO C++ standard (ISO/IEC 1998*c*). TTCN-3 does not support preprocessing wherefore only the `include` statement can be supported directly and all other preprocessing statements are not considered so far here.

The `include` preprocessor statement of IDL can be mapped onto the `import` statement of TTCN-3 to use definitions from other files. All other preprocessor statements, which are mostly

[1] Octet literals require an even number of hexadecimal digits as given by the octetstring definition

text replacements, are not matched to TTCN-3 because the IDL specification should be used after preprocessing it.

Furthermore, users could use IDL definitions by importing it and using the language option. This could look like this

```
import all from IDLDefinition language "IDL";
```

However, this is not always possible because as stated in the TTCN-3 document (ETSI 2001*a*, sec. 7.5.10) `import` allows only import of types if `module` is written in another language as TTCN! This whould prevent, for instance, use of constants.

4. IDL Specification

IDL specifications consist of *type*, *constant*, *exception*, *interface*, *module* and *value* declarations. Beside *basic* and *constructed* data types IDL provides the object oriented types `interface` and `valuetype`. This object oriented types cannot be mapped straight forward because they contain structural information which has to be considered in the TTCN configuration architecture and by the CORBA/TTCN gateway.

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.

4.1. Module Declaration

IDL uses *modules* as main grouping and scoping unit. TTCN-2 mappings use *modules* to define nested *test groups* whereas TTCN-3 owns an *module* concept which can be used to map `module` on it.

IDL	TTCN-3
`module identifier { body }`	`module identifier { body }`

4.2. Interface Declaration

Interfaces describe objects with all their access methods by using *operations* and *attributes*. Additionally, interfaces can contain local type definitions like *exceptions* and *constants* which can be used by its operations and attributes. A mapping for *interfaces* should provide a similar scoping and grouping mechanism as well as an appropriate handling under TTCN as in IDL. Because of lacking an object model in TTCN, *interfaces* have to be flattened and all *interface* definitions are stored in one `group`. Hence, import of single `interface` definitions from other *modules* via the *importing group* statement gets possible. Using `module` would not be appropriate because the module concept from IDL has to be considered as well as the module concept of TTCN.

The test configuration concept of TTCN-3 requires use of `component` and `port` resp. PCO. According the TTCN-2 mappings, interfaces are best mapped to PCOs. Hence, they are mapped to ports in TTCN-3. Hereby, components are used as a collection of interfaces resp. objects.

IDL	TTCN-3

`interface NamingContext { body }`	`group NamingContextInterface {`
	`type port NamingContext`
	`procedure { ... }`
	`}`

An `interface` defines an own type in IDL which represents the interface entity. Additionally, IDL defines an own type `object` where all interfaces are derived from. Because the port reference cannot be used inside `signature` definitions another mechanism has to be used to represent the interface reference. TTCN-2 mappings use stringified *Interoperable Object References* (IORs) or integers. TTCN-3 does not provide special support for it. In order to be consistent to CORBA stringified IOR is preferred, too.

Interfaces can make use of *inheritance* from other interfaces which can also be `abstract`. TTCN provides no object oriented concepts which provides inheritance wherefore all inherited elements have to be rolled out. Thus, they have to be handled as defined in the interface itself. In case of multiple inheritance elements have only to be inherited once!

Forward references of *interfaces* resp. *ports* can also be used in TTCN. Local interfaces require no further special mapping wherefore they can be treated as normal interfaces.

One *interface* can be instantiated many times but it can only be executed properly if the corresponding port is connected to a component. Therefore, whenever a component gets created all corresponding ports resp. interfaces resp. objects are instantiated, too. Furthermore, it is not possible to dynamically change the number of objects for a component wherefore a new component has to be created. The instantiation location depends on the component which has to be matched properly by the test system. A further discussion of this problem can be found in the TTCN-2 mappings (Mednonogov 2000, Mednonogov, Kari, Martikainen and Malinen 2000).

It makes no difference for the mapping if requested[3] or provided[4] 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 concrete matching of interface *operations*, *attributes*, *exceptions*, *types* and *constants* are described in the following sections.

4.3. Value Declaration

The `valuetype` is local like a `struct` but contains also *inheritance*, *operations* and *attributes* like an `interface` (OMG 2001, chap. 5). If `valuetype` is used as parameter it will be passed as *object-by-value* wherefore it can be seen as *interface* which is passed by value. However, `valuetype` is more similar to `struct` because only the values will be transmitted. Thus, it is mapped like a `struct` where inheritance is treated by rolling it out as it is also defined for interfaces and given operations are mapped to `external` functions. If there is only one variable for `valuetype`, it can be mapped like given by the type mapping with a special attribute, the `encode` statement, for this type as described in section 9.

[3]test system requires interface which is provided by SUT
[4]test systems provides interface which is required by SUT

The examples below show how to map `valuetype` and were used from (OMG 2001, sec 5.3, sec. 5.2.5.2).

IDL

```
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 );
};
```

TTCN-3

```
type charstring StringValue
   with { encode "IDL:valuetype
                 and IDL:string" }


group EmployeeRecordValuetype {
   type record EmployeeRecord {
      charstring name,
      charstring email,
      charstring SSN
   }

   external function
      EmployeeRecord_init(
         charstring name,
         charstring SSN );
}
```

4.4. Constant Declaration

Constant declarations can be easily transformed to TTCN-3 by use of literal (see table 1) and operator mapping for floating-point and integer values (see table 2).

Operator	IDL	TTCN-3
Unary floating-point		
positive	+	+
negative	−	−
Binary floating-point		
addition	+	+
subtraction	−	−
multiplication	*	*
division	/	/
Unary integer		
positive	+	+
negative	−	−
bit-complement	~	not4b

Operator	IDL	TTCN-3
Binary integer		
addition	+	+
subtraction	−	−
multiplication	*	*
division	/	/
modulo	%	*mod*
shift left	<<	<<
shift right	>>	>>
bitwise and	&	and4b
bitwise or	|	or4b
bitwise xor	^	xor4b

Table 2: Operators for constant expressions

IDL

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

TTCN-3

```
const IDLLong number := 9;
const IDLLong size := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;
```

5. Type Declaration

IDL provides type declarations for values and constants as seen in the sections before (see section 4.3 and 4.4) and *basic* data types, *constructor* types, *template* and *complex* types. Their mapping to TTCN-3 will be shown in the following subsections.

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.

5.1. IDL Basic Types

Mapping IDL basic data types to TTCN data types is straight forward because IDL data types are similar to ASN.1 data types which are used as data type base in TTCN, too (ITU-T 1997*a*, Open Group 2000). TTCN-3 provides more predefined types than TTCN-2 wherefore a better mapping can be provided.

Integer and Floating-Point Types Integer mapping does not differ between TTCN-2 and TTCN-3 where `integer` with range limitations are used for a proper mapping. This looks like in the following example:

TTCN-3

```
type integer IDLShort            ( -32768 .. 32767 );
type integer IDLLong             ( -2147483648 .. 2147483647 );
type integer IDLLongLong         ( -9223372036854775808 .. 9223372036854775807 );

type integer IDLUnsignedShort    ( 0 .. 65535 );
type integer IDLUnsignedLong     ( 0 .. 4294967295 );
type integer IDLUnsignedLongLong ( 0 .. 18446744073709551615 );
```

TTCN-2 has no `float` type wherefore *floats* could only be mapped onto the ASN.1 `real` type. In TTCN-3 there is now an unlimited `float` type available where `float`, `double` and `long double` from IDL can be mapped on. However, there is still no range limitation available wherefore the range limitations has to be kept in mind.

Char and Wide Char Type The IDL `char` and `wide char` type represent a single and wide character[5]. TTCN-3 introduces the character types `char` and `universal char` wherefore IDL `char` and `wchar` can now be mapped properly on it. However, TTCN uses ISO/IEC 646 as character set for char and ISO/IEC 10646 for universal char wherefore the IDL specification has to be limited to ISO/IEC 646 resp. ISO/IEC 10646 or an appropriate translation has to be used (ISO/IEC 1990, 1993).

IDL	TTCN-3
`const char letter      =  'A';` `const wchar wideLetter =  'A';`	`const char letter                 := "A";` `const universal char wideLetter := "A";`

Boolean Type The IDL `boolean` type can be mapped directly to the TTCN-3 `boolean` type.

IDL	TTCN-3
const boolean isValid = TRUE;	const boolean isValid = true;

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

IDL	TTCN-3
const octet data = 0x55;	const octetstring data = "55"H

Any Type In IDL it is possible to represent each possible IDL type with the `any` type. There is no corresponding type in TTCN available wherefore another construct is required. One solution could be using a `union` type to store all possible data types. The `union` type comprises all required types wherefore this types has to be known in advance. To prevent this limitation a `union` type for all possible types in TTCN used by the IDL mapping rules could be defined. However, it is only possible to use basic data types and well defined constructed types wherefore no generic constructed types for `set`, `record`, `union`, etc. are supported. For instance, it is not possible to provide entries for all possible kinds of `set` by using a generic `set`. Therefore, the user could define his own *restricted* any type in TTCN by using a `union` type containing only all possible types of the concrete application and not all thinkable types. This new `any` type has to be mapped to a full `any` type by the test system. However, this requires a careful handling of `any` types because type definitions could be missing.

The mapping described in (Mednonogov, Kari, Martikainen and Malinen 2000) is fixed on ASN.1 and therefore, it has problems in distinguishing types which are mapped onto the same ASN.1 type. There is no support of the `any` type given but they suggest use of the design pattern *decorator* if `any` type is used. Another mapping provides only basic types for the `any` type and let structured types open for further study (Li 1998).

TTCN-3 provides no new feature which would solve the problem of mapping the `any` type properly. Therefore, only a restricted `any` type could be used at the moment. However, using

[5]Any character set can be used for type `wide char`

any in an interface can nowadays be seen as a gap in a good system resp. interface specification resp. design. Thus, use of the any type should be prevented was is also done in ASN.1 (ITU-T 1997*a*, sec. E.3). Therefore, the any type should not occur because, if tests via TTCN are executed, the interfaces should be quite stable without use of the any type. Nevertheless, there are scenarios where the any type has to be used. An example is an application which forwards data without looking into it.

CORBA provides use of a *dynamic management of any values* but it is not appropriate to use this concept for a mapping of the any type (OMG 2001, chap. 9).

An any type for basic types could look like the one below where the *type kind* is stored in a separate variable:

TTCN data type for *CORBA any*

```
type record IDLAny {
    IDLTCKind kind,
    IDLAnyValueHelper value_ }

type union IDLAnyValueHelper {

IDLShort               short_,
IDLLong                long_,
IDLLongLong            longLong_,
IDLUnsignedShort       ushort_,
IDLUnsignedLong        ulong_,
IDLUnsignedLongLong    ulongLong_,

float                  float_,

IDLOctet               octet_,
IDLFixed               fixed_,

char                   char_,
universal char         wchar_,
charstring             string_,
universal charstring   wstring_,

boolean                boolean_,
charstring             object_  }
```

```
type enumerated IDLTCKind {
tk_null, tk_void,

tk_short, tk_ushort,
tk_long, tk_ulong,
tk_longlong, tk_ulonglong,

tk_octet, tk_boolean,
tk_float, tk_double, tk_longdouble,

tk_char, tk_wchar,
tk_string, tk_wstring,

tk_any, tk_objref,

tk_struct, tk_union, tk_sequence,
tk_enum, tk_array, tk_fixed

tk_alias, tk_except, tk_TypeCode,
tk_Principal, tk_value, tk_value_box,
tk_native, tk_none,
tk_abstract_interface  }
```

The types double and long double has to be distinguished via the *type kind* and the values have to be stored in float_.

The mappings without integer mapping are summarized in table 3. Integers are all mapped to integer with according range limitation.

5.2. Constructed Types

IDL provides the three constructed types struct, union and enum. Recursive construction of types is only permitted with the sequence template type (see section 5.3).

There is no fundamental difference between mapping to TTCN-2 and TTCN-3 for most *constructed* types given because new data types in TTCN-3 are directly introduced from ASN.1.

CORBA IDL	TTCN-2	TTCN-3
float	—	float
double	—	float
long double	—	float
boolean	BOOLEAN	boolean
octet	OCTET STRING	octetstring
char	GraphicString, IA5String(SIZE(1))	char
wchar	GraphicString, BMPString(SIZE(1))	universal char
any	CHOICE	union

Table 3: Mapping rules for basic types

Hence, only a closer mapping to this new data types gets possible. This concerns, for example, `sequence`, `sequence of`, `enumerated` and `choice` which are used as `record`, `record of`, `enumeration` and `union`. Hence, TTCN-2 mapping can mostly be used for TTCN-3, too (see table 4).

Struct `Struct` is used to collect data in one place. Because of the importance of ordering inside `struct` it is always mapped to ASN.1 `sequence`. This ordering is until now not mentioned in the IDL specification but it is indirectly mentioned in the CORBA document, for instance, in the *Internet Inter-ORB Protocol* (IIOP) specification and *type code* specification (OMG 2001, chap. 15). Therefore, mapping onto the new data type `record` in TTCN-3 can be used.

IDL	TTCN-3

`typedef struct NC {`
`string id;`
`string kind;`
`} NameComponent;`
	`type record NameComponent {`
	`string id,`
	`string kind`
	`}`

Discriminated Unions *Unions* can store different types in one place but only one at the same time. Under IDL `union` is *discriminated* wherefore the actual type has to be specified, too. They can be mapped directly to the TTCN-3 `union` type where the type information is not necessary. Nevertheless, TTCN-3 provides the function `isChosen()` to get the actual type of the `union` variable.

IDL	TTCN-3

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

Enumerations In both languages *enumerations* can be used on the same way.

IDL

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

TTCN-3

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

CORBA IDL	TTCN-2/ASN.1	TTCN-3
struct	SEQUENCE	record
enum	ENUMERATED	enumeration
union	SEQUENCE	record

Table 4: Mapping rules for constructed types

5.3. Template Types

IDL supports the template types `sequence`, `string`, `wide string` and *fixed* type. Their mapping is summarized in table 5.

Sequence IDL `sequence` is used to provide support of one-dimensional arrays with a fixed maximum size and a length. In contradiction to fixed arrays (see section 5.4) only the valid sequence entries will be transmitted and the empty entries will not and therefore, use of unbounded sequences is possible, too. This makes a matching onto arrays in TTCN impossible wherefore only `set of` and `record of` are available. To keep the ordering of sequences the type `record of` has to be chosen to provide an appropriate matching to TTCN.

IDL

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

TTCN-3

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

String and wstring `string` and `wstring` are *sequences* of `char` and `wchar`. In TTCN-2, `characterstring` uses several different predefined types which are all replaced in TTCN-3 by two `characterstring` types which use ISO/IEC 646 and ISO/IEC 10646 character encoding. Therefore, `string` and `wstring` have to be mapped to `charstring` and `universal charstring`.

IDL

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

TTCN-3

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

Fixed Types The `fixed` type represents a fixed-point decimal number. There is no corresponding type for `fixed` type in TTCN available whereby a new type has to be created. Because of the similarities between TTCN and C the solution from the C language mapping of IDL (OMG 1999, section 1.14) is used. It maps the `fixed` type to a `struct` which stores the number in a *char array* and the digit and scale number in extra variables. In TTCN-3 it can be realised via a `record` and a `charstring` to store the number. A `record` is preferred against a `set` because of the initialiser notation for records which makes use of `fixed` types under TTCN-3 more convenient.

IDL	TTCN-3

`typedef fixed<12,7> Fix;`	`type record IDLFixed {`
	`    IDLUnsignedShort digits,`
	`    IDLShort         scale,`
	`    charstring       value_ }`
	
	`var IDLFixed fix :=`
	`     { 12, 7, "12345.1234567" };`

See the definition of `IDLUnsignedShort` and `IDLShort` on page 10.

CORBA IDL	TTCN-2/ASN.1	TTCN-3
sequence	SEQUENCE OF	record of
string	GraphicString, IA5String	charstring
wstring	GraphicString, BMPString	universal charstring
fixed	—[1]	record

Table 5: Mapping rules for template types

5.4. Complex declarator

The last kind of type declarator are the complex types `array` and `native`.

Arrays IDL `array` can be mapped directly to the TTCN `array` type because they provide the same functionality.

IDL	TTCN-3
`typedef long NumberList[100];`	`var IDLLong NumberList[100];`

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

IDL	TTCN-3
`native MyNativeVariable;`	`address MyNativeVariable;`

[1]Mapping of this type was not considered.

6. Exception Declaration

In IDL `exception` is used in conjunction with operation resp. procedure calls to handle exceptional conditions during performing an operation call. Therefore, a special *struct* like `exception` type is provided which has to be associated with each operation which can raise this exception. TTCN-3 supports also the use of exceptions with procedure calls by binding it to `signature` definitions but provides no special `exception` type wherefore *exceptions* are defined like `struct` by use of `record`. TTCN-2 has no support of *exceptions* thus it is realized by using PCOs with asynchronous communication to get exception information from the test system.

The part exception *definition* is shown in the following example and use of exception binding in signature definitions and exception *catching* is shown in context of operation declaration in the section 7 on page 18.

IDL

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

TTCN-3

```
// 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 par1, Name name) := {
    why := missing_node,
    rest_of_name := name }
```

7. Operation Declaration

Beside *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 via the synchronous communication operations which are used in conjunction with ports and components as stated earlier. *Operations* under IDL consist of *invocation* semantics, *return* results, *identifier*, *parameter* list, optional `raise` expression and optional `context` expression. The matching of all this parts to TTCN-3 will be described now.

Operation Attribute IDL supports an optional `oneway` attribute which implies best-effort invocation semantics without a guarantee of delivery but with a most-once invocation semantics. `oneway` operations have to provide no `out` parameters and the return type `void`. Furthermore, no `raise` expressions are allowed but standard exceptions can still be raised. If no attribute is given the invocation semantic is at-most-once in case of an exception and exactly-once if it returns successfully.

Message or *procedure* based ports could be used for `oneway` procedures because both would be a valid mapping from IDL point of view. However, use of *procedure* based ports for `oneway` procedures is recommended because the IDL specification makes no guarantee that `oneway` calls

are non-blocking or asynchronous. Furthermore, CORBA implements oneway procedures via synchronous communication, too. Hence, best mapping of both kind of operations is given by use of synchronous communication wherefore no distinction for oneway procedures gets necessary. However, it is not the same for CORBA wherefore the IDL information in the interface repository can be used to detect oneway operations and to handle them appropriate. Furthermore, use of the extension statement could be used to mark oneway operations what should be preferred.

By the way, CORBA supports also the *Asynchronous Method Invocation* (AMI) to provide non-blocking requests. It is realized by a client-side mapping which makes no or only small server side changes necessary. Therefore, special methods will be produced in addition to provide the new functionality. This new methods use still the synchronous communication mechanism wherefore no modifications in IDL have been necessary. However, there was a new IDL introduced called *implied IDL* which is generated from the IDL specification with the additional operations for the client side and is used to generate the client implementation stubs.

Introduction of AMI leads to the fact that a client is also a server if the *callback* model is used. This model requires a reply handler which is invoked by the server. Hence, mapping of IDL to TTCN-3 gets necessary from client and server perspective. Therefore, CORBA exceptions, as defined in the section before, can also be raised by the tester.

Parameter Declarations 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 exactly the same semantics.

return, out and inout parameters of operations have to be catched by a getreply statement in TTCN-3 which should be given in the call context. IDL makes the following restriction for passing parameters:

> When an unbounded **string**, **wstring** or **sequence** is passed as an **inout** parameter, the returned value cannot be longer than the input value (OMG 2000).

Hence, this has to be kept in mind by writing test cases and by the test environment.

Raises Expressions A raise expression specifies all raisable exceptions by an operation and can be mapped directly to TTCN-3 because it can be given by the procedure signature definition by specifying an exception. Nevertheless, each operation can raise a standard exception.

Context Expressions context expressions provide access to local properties for the called operation. This properties consist of a name and a string value. As used in (OMG 2001, sec. 4.6) the context expression can be matched by redefining the operation with the context parameters included into the operation parameters. This is done in a TTCN-2 mapping introducing an additional array parameter, too (Mednonogov 2000, Mednonogov, Kari, Martikainen and Malinen 2000). The additional parameter should consist of a sequence containing for each context parameter a struct containing two string variables for the context name and value.

IDL

```
// 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-3

Operation definition

```
type record IDLContextElement {
    charstring name,
    charstring value_
}

type record of IDLContextElement IDLContext;

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

signature RemoteProcSignature2(
             in IDLLong Par21, in IDLLong Par22,
             in charstring name2 )
          with { extension "IDL:oneway" };



type port RemoteProcPort procedure {
          out RemoteProcSignature1;
          out RemoteProcSignature2
}

type component CorbaSystem {
     port RemoteProcPort PCO
}
```

Operation usage

```
// not found exception is defined in section ‘‘exception declaration’’

var IDLContextElement contextElement := {
        name := "MyContext1", value_ := "MyContextValue" };
var IDLContext idlContext := { contextElement };
```

```
template RemoteProcSignature1 RemoteProcTemplate1 := {
    Par11 := 0,
    Par12 := 1,
    name1 := "my name",
    context := idlContext
}

template RemoteProcSignature2 RemoteProcTemplate2 := {
    Par21 := 2,
    Par22 := 3,
    name2 := "my other name"
}

var CorbaSystem myCorbaSystem := CorbaSystem.create;
connect(self:MyPCO, MyCorbaSystem:PCO);
myCorbaSystem.start;

MyPCO.call( RemoteProcTemplate1 ) {
    [] MyPCO.getreply(RemoteProcSignature1:{-,*,*} value *) -> value MyResult1
                      param(MyPar12, MyName1) sender MySender1 {}
    [] MyPCO.catch( RemoteProcSignature1,
                   MyNotFoundExceptionTemplate ) {
           verdict.set(fail);
           stop;
       }
}

MyPCO.call( RemoteProcTemplate2 );

// raising an exception can be done on this way but it is not used
// because only the SUT should raise this exception!!!
var NotFoundException myNotFoundException := {
 why          := missing_node,
 rest_of_name := "noname"

}

MyPCO.raise( RemoteProcSignature1, myNotFoundException );
```

8. Attribute Declaration

An `attribute` is like a *set* and *get* operation pair to access a value. If an attribute is `readonly` only the *get* operation has to be provided! Therefore, attribute mapping is given by the operation mapping.

IDL

```
attribute string object_type;
```

TTCN-3

```
signature RemoteAttribGetSignature() return charstring;
signature RemoteAttribSetSignature( in charstring object_type );

type port RemoteProcPort procedure {
     out RemoteAttribGetSignature;
     out RemoteAttribSetSignature
}

type component CorbaSystem {
     port RemoteProcPort PCO
}

var CorbaSystem MyCorbaSystem := CorbaSystem.create;
connect(self:MyPCO, MyCorbaSystem:PCO);
CorbaSystem.start();

// get
MyPCO.call() {
    [] MyPCO.getreply( value * ) -> value MyResult {}

       // catch all exceptions
    [] MyPCO.catch {
           verdict.set( fail );
           stop;
       }
}

// set
template RemoteAttribSetSignature RemoteAttribSetSignatureTemplate := {
    object_type := "my name"
}

MyPCO.call( RemoteAttribSetSignatureTemplate );
}
```

9. 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 wherefore the IDL scoping rules have to be mapped appropriate to allow a 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 wherefore 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` name as designated in IDL and TTCN. The use of `module` names is not necessary because they are reflected by the TTCN `module` structure. As separator a underscore shall be used and existing underscores shall be doubled. Use of path names was also suggested by the TTCN-2 mappings.

To indicate the special treatment of TTCN statements derived from IDL, TTCN-2 mappings are using special naming schemes. This simplifies coding and type differentiation. TTCN-3 provides a new mechanism to attach attributes to language elements. Use of this attributes make code more readable and require no special naming scheme. The `encode` attribute is used to define encoding rules to type definitions. Therefore, the `encode` attribute can be used to indicate the derivation of types from IDL and the special treatment for encoding by the test system resp. CORBA/TTCN gateway. For example, this could look like

```
type integer IDLShort ( -32768 .. 32767 ) with { encode "IDL:short" };

type record NameComponent {
    MyString id,
    MyString kind } with { encode "IDL:struct" };
```

This requires an own naming scheme to tag all types. It is suggested to use the type name and if necessary add special attributes like `valuetype` via the keyword *and*.

The `encode` attribute can only be used for type definitions and therefore, the `extension` attribute has to be used for other language elements like `signature` definitions. It is suggested to use the IDL keyword which represents the special handling like `oneway` for `signature` definitions. Several attributes shall be concatenated via the keyword *and* as used for `encode` attribute.

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

10. Conclusion

This report provides a mapping of IDL to TTCN-3 where rules are given and explained to map elements. It is not always possible to provide satisfiable mappings because mapping for the `any` type is not really solved. Also, mapping for `interface` and `valuetype` is not powerful enough in order to use all IDL features in a proper way. However, this is no limitation in using it but for convenience. The use of ASN.1 to map type `any` and to use inheritance seems to be interesting which is left open for further study.

Furthermore, some mappings could not be very exactly because of lacking support in TTCN-3, for instance, `double` and `long double`. The latter ones can only be matched to `float` which has no exact size definitions. *Operations*, *attributes* and *exceptions* can be mapped well to TTCN-3 because synchronous communication with an IDL like syntax and semantics was introduced especially for this case.

CORBA specific parts like variable handling if used as `inout` parameters are not considered in TTCN because they should be done by the compiler or interpreter and/or the underlying test system. Nevertheless, if IDL is not precise enough, e.g., the `oneway` attribute for operations, the CORBA specification was used as reference.

There was no discussion about technical implementation details. However, projects based on TTCN-2 have shown that it is possible to use TTCN-2 and CORBA together. Furthermore, TTCN-3 gives support to synchronous communication wherefore implementing test suites gets easier, too. Nevertheless, introduction of AMI in CORBA requires testing of the asynchronous and synchronous functionality of the SUT.

If TTCN-3 is used to test CORBA systems, the language mapping of TTCN-3 should be equivalent to the IDL language mapping.

To sum up, in contrast to TTCN-2, mapping to TTCN-3 is more comfortable and powerful. However, the report has not considered the concrete mapping for compilers and test environments to existing systems. Hence, this has to be done later for each desired application area like CORBA systems.

A. IDL Concept Mapping List

Table 6 lists the mapping of keywords and concepts of IDL to TTCN-3 keywords or concepts. Literal and operator mapping can be seen in section 2 and 4.4.

IDL	TTCN-3
`FALSE`	`false`
`Object`	`charstring`
`TRUE`	`true`
`abstract`	has to be rolled out
`any`	`record, union, enumerated`
`array`[1]	`array`
`attribute`	get (and set) operation
`boolean`	`boolean`
`char`	`char`
`const`	`const`
`context`	additional procedure parameter with own type
`enum`	`enumerated`
`exception`	`record`
`fixed`	`record` with integers and charstring
`float, double, long double`	`float`
`in`	`in`
`inout`	`inout`
`interface`	`group, port`
`local`[2]	—
`long`	`integer`[3]
`long long`	`integer`[3]
`module`	`module`
`native`	`address`
`octet`	`octetstring`
`oneway`	operation with `encode` attribute
operation[1]	`signature` for procedure
`out`	`out`
`raises`	`exception`
`readonly`	only a get operation for the attribute

[1] This is a IDL concept and not a keyword

[2] This keyword requires no special treatment because it makes no difference for its corresponding statement. The difference has to be made by the user, used tools and/or test environments.

[3] `integer` with according range limitation

IDL	TTCN-3
sequence	record of
short	integer[1]
string	charstring
struct	set
typedef	type
union, switch, case	union
unsigned long	integer[1]
unsigned long long	integer[1]
unsigned short	integer[1]
valuetype	record, function (or directly)
wchar	universal char
wstring	universal charstring

Table 6: Conceptual List of IDL Mapping

[1] integer with according range limitation

B. Derived TTCN-3 data types

Table 7 lists the derived TTCN-3 data types. They can be used if a direct mapping to a TTCN-3 data type was not possible. Nevertheless, it is also possible to define special TTCN data types for all IDL data types.

IDL	TTCN-3 (basic)	TTCN-3 (derived)
any	record, union, enumerated	IDLAny, IDLAnyValueHelper, IDLTCKind
fixed	record with integers and charstring	IDLFixed
long	integer[1]	IDLLong
long long	integer[1]	IDLLongLong
octet	octetstring	IDLOctet
short	integer[1]	IDLShort
unsigned long	integer[1]	IDLUnsignedLong
unsigned long long	integer[1]	IDLUnsignedLongLong
unsigned short	integer[1]	IDLUnsignedShort
context[2]	--[3]	IDLContextElement, IDLContext

Table 7: Derived TTCN-3 data types

[1] integer with according range limitation
[2] context is no data type
[3] Mapping of context was not considered.

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

Table 8 lists a comparison of IDL, ASN.1, TTCN-2 and TTCN-3 data types and is based on the documents (Li 1998, Mednonogov 2000, Mednonogov, Kari, Martikainen and Malinen 2000, Open Group 2000).

Table 8: 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	charstring
any	ANY DEFINED BY[3] or SEQUENCEtypecode, anyValue	CHOICE	record, union, enumerated
array	SEQUENCE OF (with sizeConstraint subtype)	SEQUENCE SIZE(n) OF	array
boolean	BOOLEAN	BOOLEAN	boolean
char	GraphicString	GraphicString or IA5String(SIZE(1))	char
enum	ENUMERATED	ENUMERATED	enumerated
exception	SPECIFIC ERRORS	SEQUENCE	record
fixed	—[1]	—[1]	record with integers and charstring
float, double, long double	REAL	—[1]	float
long	INTEGER	INTEGER	integer[2]
long long	INTEGER	INTEGER	integer[2]
native type	—[1]	—[1]	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	integer[2]
string	GraphicString	GraphicString	charstring
struct	SEQUENCE	SEQUENCE	record
union, switch, case	CHOICE (with ASN.1 TAGS)	SEQUENCE	union
unsigned long	INTEGER	INTEGER	integer[2]
unsigned long long	INTEGER	INTEGER	integer[2]
unsigned short	INTEGER	INTEGER	integer[2]
valuetype	—[1]	—[1]	record, function (or directly)
wchar	—[1]	GraphicString or BMPString(SIZE(1))	universal char
wstring	—[1]	GraphicString	universal charstring

[1] Mapping of this type was not considered.
[2] integer with according range limitation
[3] superseded in ASN.1 from 1997 (ITU-T 1997*a*)

D. Complete Example

The following example shows how a mapping would like if a complete IDL and TTCN-3 specification, inclusive a test case, is used. It is only intended to give an impression 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.

D.1. 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 =  'A';

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

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

D.2. TTCN-3

```
// ************************************
```

```
// Module with General IDL Declarations
// **********************************

module IDLDefault {
    // ************
    // Integer Type
    // ************
    type integer IDLShort    (-32768 .. 32767) with { encode "IDL:short" };
    type integer IDLLong     (-2147483648 .. 2147483647) with { encode"IDL:long" };
    type integer IDLLongLong (-9223372036854775808 .. 9223372036854775807)
                             with { encode "IDL:longlong" };

    type integer IDLUnsignedShort    ( 0 .. 65535 ) with { encode "IDL:ushort" };
    type integer IDLUnsignedLong     ( 0 .. 4294967295 ) with { encode "IDL:ulong" };
    type integer IDLUnsignedLongLong ( 0 .. 18446744073709551615 )
                                         with { encode "IDL:ulonglong" };

    type octetstring IDLOctet with { encode "IDL:octet" };

    // **********
    // Fixed Type
    // **********
    type record IDLFixed {
        IDLUnsignedShort digits,
        IDLShort         scale,
        charstring       value_ } with { encode "IDL:fixed" };

    // ********
    // Any Type
    // ********
    type union IDLAnyValueHelper {

        IDLShort               short_,
        IDLLong                long_,
        IDLLongLong            longLong_,
        IDLUnsignedShort       ushort_,
        IDLUnsignedLong        ulong_,
        IDLUnsignedLongLong    ulongLong_,

        float                  float_,

        IDLOctet               octet_,
        IDLFixed               fixed_,

        char                   char_,
        universal char         wchar_,
        charstring             string_,
        universal charstring   wstring_,

        boolean                boolean_,
        charstring             object_
    }


    type enumerated IDLTCKind {

        tk_null, tk_void,

        tk_short, tk_ushort,
```

```
        tk_long, tk_ulong,
        tk_longlong, tk_ulonglong,
        tk_octet, tk_fixed,

        tk_float, tk_boolean,
        tk_double, tk_longdouble,

        tk_char, tk_wchar,
        tk_string, tk_wstring,

        tk_any, tk_objref,

        tk_struct, tk_union,
        tk_sequence, tk_array,
        tk_enum,

        tk_alias, tk_except,

        tk_TypeCode, tk_Principal,
        tk_value, tk_value_box,
        tk_native,
        tk_abstract_interface,

        tk_none
    }


    type record IDLAny {
        IDLTCKind kind_,
        IDLAnyValueHelper value_
    } with { encode "IDL:any" };

    // ************
    // Context Type
    // ************
    type record IDLContextElement {
       charstring name,
       charstring value_
    }

    type record of IDLContextElement IDLContext with { encode "IDL:context" };
} // end of module IDLDefault


// **************************
// Module with TTCN-3 Example
// **************************

module ttcnExample {

    // **************************
    // Import All from IDLDefault
    // **************************
    import all from IDLDefault;

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

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

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

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

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

type charstring MyString with { encode "IDL:string" };

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

// ******
// Struct
// ******
type record NameComponent {
    MyString id,
    MyString kind
} with { encode "IDL:struct" };

// *****
// Union
// *****
type union MyUnion  {
    boolean b,
    charstring c,
    IDLOctet o,
    IDLShort s
} with { encode "IDL:union" };

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

// ********
// Sequence
// ********
type record of NameComponent Name with { encode "IDL:sequence" };

type record of NameComponent Key with { encode "IDL:sequence" };

//******
// Fixed
// *****
// see using of fixed in testcase below
```

```
// ******************
// Complex Declarator
// ******************
// see using of array in testcase below

// see using of native in testcase below



// ********************
// Valuetype Definition
// ********************
type charstring StringValue
    with { encode "IDL:valuetype and IDL:string" };

group EmployeeRecordValuetype {
    type record EmployeeRecord {
        charstring name,
        charstring email,
        charstring SSN
    } with { encode "IDL:valuetype and IDL:struct" };

    external function EmployeeRecord_init(charstring name, charstring SSN);
}

// ********************
// Interface Definition
// ********************
group NamingContextInterface {
    // attribute object_type
    signature ObjectTypeGetSignature() return charstring;
    signature ObjectTypeSetSignature( in charstring 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 par1, Name name ) := {
            why          := missing_node,
            rest_of_name := name
        }

    //
    // bind procedure
    //
```

```
    signature BindSignature( in Name n, inout octetstring obj,
                             inout octetstring  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 charstring obj )
         with { extension "IDL:oneway" };


    template RebindSignature RebindTemplate ( charstring 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 MyMTC be defined


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

    // examples to show how above defintions 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" };

//
// Array
//
var integer numberList[100];

//
// Native
//
var address MyNativeVariable;


//
// Procedure Calls
//
var charstring myResult1;
var Key        myResult2;
var MyString   myResult3;
var charstring 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 )
            {
                verdict.set(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
```

References

ETSI. 2001*a*. Methods for Testing and Specification (MTS) — The Tree and Tabular Combined Notation version 3 — Part 1: TTCN-3 Core Language. European Standard ETSI ES 201 873-1 European Telecommunications Standards Institute Sophia-Antipolis, France: .

ETSI. 2001*b*. Methods for Testing and Specification (MTS) — The Tree and Tabular Combined Notation version 3 — Part 2: TTCN-3 Tabular Presentation Format. European Standard ETSI ES 201 873-2 European Telecommunications Standards Institute Sophia-Antipolis, France: .

ISO/IEC. 1990. Information Technology — ISO 7-bit coded character set for information exchange. International Standard 646 ISO/IEC.

ISO/IEC. 1993. Information Technology — Universal Multiple Octet-Coded Character Set (UCS). International Standard 10646 ISO/IEC.

ISO/IEC. 1998*a*. Information Technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1. International Standard 8859-1 ISO/IEC.

ISO/IEC. 1998*b*. Information Technology — Open Systems Interconnection — Conformance Testing Methodology and Framework — Part 3: The Tree and Tabular Combined Notation (Second Edition). International Standard 9646-3 ISO/IEC.

ISO/IEC. 1998*c*. Information Technology — Programming Languages — C++. International Standard 14882 ISO/IEC.

ISO/IEC. 1999. Information Technology — Open Distributed Processing — Interface Definition Language. International Standard DIS 14750 ISO/IEC.

ITU-T. 1997*a*. Recommendation: Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. International Standard X.680 ITU-T.

ITU-T. 1997*b*. Recommendation: Information Technology — Open Distributed Processing — Interface Definition Language (IDL). International Standard X.920 ITU-T.

J. Grabowski, A. Wiles, C. Willcock and D. Hogrefe. 2000. On the Design of the new Testing Language TTCN-3. In *Proceedings of the IFIP TC6/WG6.1* 13th *International Conference on Testing of Communicating Systems (TestCom 2000), August 29 – September 1, 2000, Ottawa, Canada*, ed. H. Ural, R.L. Probert and G.v. Bochmann. IFIP – The International Federation for Information Processing Kluwer Academic Publishers.

Leach, E. 2000. Enhanced Techniques for CORBA Validation CORVAL2 – Validating Multi-Vendor CORBA Conformance and Interoperability in Heterogeneous Environments. D29 – white paper The Open Group. European Commission Project Number IST-1999-11131.
URL: *http://www.opengroup.org/corval2*

Li, M. 1998. Testing Computational Interfaces of CORBA Service using TTCN and CORBA. Diplomarbeit Fachgebiet Telekommunikationsnetze, Fachbereich Elektrotechnik, Technische Universität Berlin, Germany.

Mednonogov, A. 2000. Calypso Gateway specification, version 0.07. Technical report Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology, Finland.

Mednonogov, A., H.H. Kari, O. Martikainen and J. Malinen. 2000. Conformance Testing of CORBA Services using Tree and Tabular Combined Notation. In *Proceedings of the IFIP TC6/WG6.1* 13th *International Conference on Testing of Communicating Systems (TestCom 2000), August 29 – September 1, 2000, Ottawa, Canada*, ed. H. Ural, R.L. Probert and G.v. Bochmann. IFIP – The International Federation for Information Processing Kluwer Academic Publishers pp. 193–208.

OMG. 1999. C Language Mapping Specification. OMG Formal Document FORMAL/99-07-35 Object Management Group.

OMG. 2000. The Common Object Request Broker — IDL Syntax and Semantics. OMG Formal Document FORMAL/00-10-07 Object Management Group. Version 2.4.

OMG. 2001. The Common Object Request Broker — Architecture and Specification. OMG Formal Document FORMAL/2001-02-01 Object Management Group. Version 2.4.2.

Open Group. 2000. Inter-Domain Management: Specification & Interaction Translation. Technical Standard C802 Open Group.
URL: *http://www.jidm.org*

Schieferdecker, I. and J. Grabowski. 2000. "Conformance Testing with TTCN." *Telektronikk — Languages for Telecommunication Applications* 96(4):85–95.

Schieferdecker, I., M. Li and A. Hoffmann. 1998. Conformance Testing of TINA Service Components — The TTCN/CORBA Gateway. In *Proceedings of the* 5th *International Conference on Intelligence and Services in Networks, IS&N'98, Antwerp, Belgium, May 25–28, 1998*, ed. S. Trigila, A. Mullery, M. Campolargo, H. Vanderstraeten and M. Mampaey. Vol. 1430 of *Lecture Notes in Computer Science* Springer pp. 393–408.