Helsinki University of Technology

Espoo, Finland

CALYPSO GATEWAY SPECIFICATION ver. 0.07

Author:	Alexey Mednonogov
Date:	28-JAN-2000

CONTENTS

1. Introduction

1.1. Architecture of Abstract Test Suite

We view SUT as a collection of CORBA objects. Each CORBA object may act either as a Server, or as a Client, or both (Mixed), or as a Complex object that incorporates zero or more Servers and zero or more Clients. Servers, Clients and Mixed objects are viewed as special cases of Complex objects.

Mixed and Complex objects are viewed as a composition of several Clients and several Servers. Hence, hereafter only interoperation of ATS with Clients and Servers is specified. ATS designer must logically decompose Mixed and Complex objects into Servers and Clients and then follow the specification.

One distinct type of objects is Factory which dynamically creates new CORBA objects. Such objects will be later referred to as Product objects.

In the run of the whole specification we refer only to Clients, Servers and Products visible outside SUT. SUT may have its own internal Servers, Clients and Products not observable by ATS. These are left out of consideration.

Each Server, Client and Product must be associated with its own PCO. Each PCO must be associated with its own object, either Client, Server or Product, or it may be defined by ATS either for its internal purposes (e.g. as a CP) or for testing non-CORBA objects.

Mixed and Complex objects normally communicate with ATS via several PCOs.

All Servers are up and running before execution of Test Case starts and their location in the CORBA Universe is known in advance.

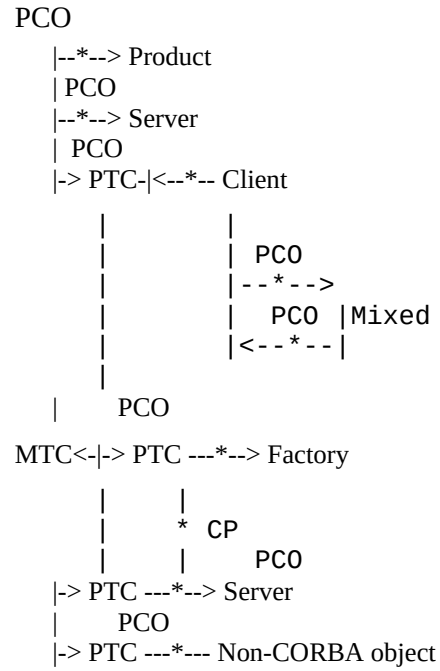
Clients are connected to ATS by an operator manually at ATS run-time once the appropriate instruction is issued by the ATS. By the moment of sending such an instruction, the Gateway is ready to supply location of Universal Servant to the Client. For more details about Universal Servants, see section 2.3.

All Servers, Universal Servants and Products must be registered within Interface Repository. This shall be done by providing appropriate IDL descriptions to the IR.

One or more PCOs may be grouped within a Test Component (TC). No rules or restrictions are defined for grouping the PCOs, and it completely depends on ATS designer. Moreover, details describing creating TCs and means of communication between them are out of scope of this specification.

The sample architecture of ATS is shown on Fig. 1.

Figure 1. Sample architecture of ATS:



1.2. Grey-Box Testing Approach

The fact that we view Mixed and Complex objects as a composition of Servers and Clients lead us to a “grey-box” testing approach, i.e. minor modifications to the code may be needed in order to initiate a normal communication of SUT with ATS.

Consider the following example: Mixed CORBA object under test will advertise its server-side IOR to the world only after it obtains IOR of the server that will handle its client-side requests. In this case modifications to the code aimed to change this situation will be needed since it is specified that all server-side IORs of SUT must be known BEFORE Test Case starts execution, but Test Case will be able to provide IORs that handle client-side requests of the SUT only AFTER it starts execution. Thus, Mixed object must prepare its own server-side IOR beforehand and wait until server-side IOR provided by ATS appears.

It may also be needed for operator to have some means to manually invoke Client requests, whenever a corresponding IMPLICIT SEND operation is defined in ATS as described in section 10.

1.3. Abbreviations

Throughout the whole specification, the following abbreviations are used:

(a) CORBA-specific abbreviations:

CORBA - Common Object Request Broker Architecture

```
DSI    - Dynamic Skeleton Interface
IDL    - Interface Definition Language
IOR    - Interoperable Object Reference
OMG    - Object Management Group
ORB    - Object Request Broker
```

(b) TTCN-specific abbreviations:

```
ASN.1    - Abstract Syntax Notation One
ATS      - Abstract Test Suite
CP       - Coordination Point
LT       - Lower Tester
MTC      - Main Test Component
PCO      - Point of Control and Observation
PDU      - Protocol Data Unit
PIXIT    - Protocol Implementation eXtra
           Information for Testing
PTC      - Parallel Test Component
SUT      - System Under Test
TC       - Test Component
TTCN     - Tree and Tabular Combined Notation
```

TTCN.MP - TTCN, Machine Processable form

1.4. Prefixes

In consequent sections, various prefixes for identifiers will be introduced, aimed at providing non-conflicting name spaces for semantically distinct units. To facilitate referencing, here all these prefixes are enumerated and briefly explained. For further details, see corresponding sections:

```
x     - prefix for PIXIT parameter names
v     - prefix for Test Case variable names

c     - prefix for constraint names
p     - prefix for PDU names
```

PCO - prefix for PCO names

PAR - prefix for Call/Reply PDU fields
containing operation parameters

RET - prefix for Call/Reply PDU fields
containing return value of operation

IDL - prefix for IDL simple type names
mapped to TTCN

TYPE - prefix for IDL complex type names
mapped to TTCN

```
GW    - prefix for Gateway-specific type
        definitions mapped to TTCN
```

enum - prefix for IDL enumerated values
mapped to TTCN

seq - prefix for field identifiers of IDL
structured types mapped to ASN.1 SEQUENCE

```
ch    - prefix for field identifiers of IDL
        union types mapped to ANS.1 CHOICE
```

Recommended prefixes (not used in the current specification. However, their use is encouraged whenever needed):

```
t     - prefix for timer identifier
tc    - prefix for Test Case identifier
ts    - prefix for Test Step identifier
```

2. Registering Client and Server Objects

In order to enable invocation of CORBA operations within ATS, physically installed CORBA objects must be associated with their logical representation inside ATS beforehand by means of registration process. Usually, registration is done in the preamble of a Test Case. Please note that registering Factory object is similar to registering Servers or Clients. Registration of Product objects is covered in section 12.

2.1. Mapping of IDL Interface to TTCN PCO

ATS designer must know in advance how many CORBA objects he is going to register or create dynamically. He must know also which IDL interface declaration describes each object. Each object must be defined then as a PCO in PCO declarations part of ATS. Rules for declaring a PCO are as follows:

(a) PCO Name ($PCO_Id in MP notation) must begin with the prefix:

["PCO"] [<Instance_Id>] ["_i"]

Here <Instance_Id> is a decimal number aimed to distinguish between different instances of one IDL interface. Note that even if you define the only instance corresponding to a single IDL interface, you are still not eligible to omit <Instance_Id>. Instances are enumerated in ascending order starting from 1.

(b) For each consequent declaration of nested module in IDL file the following construct is appended to the end of the $PCO_Id (<Module_Id> is a name of the module):

[<Module_Id>] ["_i"]

(c) Interface identifier completes definition of $PCO_Id by the following construct (<Interface_Id> is a name of the interface):

[<Interface_Id>]

(d) Underscore characters ("_") in names of modules and interfaces are doubled in definition of $PCO_Id.
(e) PCO Type Identifier ($PCO_TypeId) must be CORBA_PCO both for Server and Client objects.
(f) PCO role ($PCO_Role) is always defined as "Lower Tester" ("LT").

NOTE: Module/interface names are separated from each other by "_i" sequence, and all occurences of "_" characters in module/interface names are duplicated. This is done to make impossible such situation when different name scopes are mapped to the same PCO name. Here a well-known design pattern called "bit stuffing" is utilized. To remind the story, low-level network protocols use bit stuffing to mark end of transmission, so that boundaries of packets are unambiguously recognized. Bit stuffing is introduced here in order to mark boundaries of module names. The same approach will be used throughout the whole specification as needed.

Here are several examples of IDL-to-TTCN mapping for modules and interfaces illustrating concept of bit stuffing:

```
// IDL: Declaration of interface "ModA.ModB.IntC":
module ModA {
        module ModB {
interface IntC {
                ...
        };
    };
};
```

// TTCN.MP: Declaring PCOs for two instances of interface "ModA.ModB.IntC":

$Begin_PCO_Dcls

...
$PCO_Dcl

$PCO_Id PCO1_iModA_iModB_iIntC
$PCO_TypeId CORBA_PCO
$PCO_Role LT
$Comment /* First instance of "IntC" */

```
    $End_PCO_Dcl
    $PCO_Dcl
    $PCO_Id PCO2_iModA_iModB_iIntC
    $PCO_TypeId CORBA_PCO
    $PCO_Role LT
    $Comment /* Second instance of “IntC” */
    $End_PCO_Dcl
    . . .
$End_PCO_Dcls
```

```
// IDL: Declaration of interface “ModA_iModB.IntC”:
module ModA_iModB {
        interface IntC {
            . . .
    };
};

// TTCN.MP: Declaring PCO for the only instance of interface “ModA_iModB.IntC”:
$Begin_PCO_Dcls
    . . .
    $PCO_Dcl
    $PCO_Id PCO1_iModA__iModB_iIntC
    $PCO_TypeId CORBA_PCO
    $PCO_Role LT
    $Comment /* Instance of “ModA_iModB.IntC” */
    $End_PCO_Dcl
    . . .
$End_PCO_Dcls
```

```
// IDL: Declaration of interfaces “ModA__._ModB.IntC” and “ModA_.__ModB.IntC”:
module ModA__ {
        module _ModB {
interface IntC {
                . . .
            };
    };
};

module ModA_ {
        module __ModB {
interface IntC {
                . . .
            };
    };
};

// TTCN.MP: Declaring PCOs for instances of “ModA__._ModB.IntC” and // “ModA_.__ModB.IntC”:
$Begin_PCO_Dcls
    . . .
    $PCO_Dcl
    $PCO_Id PCO1_iModA_____i__ModB_iIntC
```

```
    $PCO_TypeId CORBA_PCO
    $PCO_Role LT
    $Comment /* Instance of "ModA__._ModB.IntC" */
    $End_PCO_Dcl
    $PCO_Dcl
        $PCO_Id PCO1_iModA___i____ModB_iIntC
        $PCO_TypeId CORBA_PCO
        $PCO_Role LT
    $Comment /* Instance of "ModA_.__ModB.IntC" */
    $End_PCO_Dcl
    . . .
$End_PCO_Dcls
```

2.2. Registering Server Objects

2.2.1. Declaring PCOs for Server Objects

PCOs for Server objects must be declared exactly as defined in section 2.1. IDL description of a Server object is taken as a base for constructing PCO Name.

2.2.2. Declaring Location of Server Objects

Location of Server objects in the CORBA Universe must be known before the execution of ATS starts, and this must be known either as IOR (Form 1), or as IOR file location (Form 2), or as location of Server object in Naming Service (Form 3).

Note for Form 1 that as soon as host name, port number or object key used by Server is changed, IOR changes and appropriate change may be needed to be done in PIXIT proforma. Thus, it may be wise to have these parameters fixed as long as possible. E.g. in ORB ORBacus you may launch your objects with command-line params "-OAhost <host>" and "-OAport <port>" for this purpose, and use Named Servants in order to have object key fixed.

Note for Form 2 that file path will be interpreted by Gateway, so declaring e.g. "hello.ref" will force Gateway to search for this file in Gateway's current directory.

Actual value of IOR, or IOR file location, or position in Naming Service, is defined outside ATS in PIXIT proforma in OMG-specific manner. ATS itself only contains references to these external parameters in section "Parameter Declarations". These must adhere to the following format:

(a) Parameter Name ($TS_ParId) must follow the same rules as defined for the corresponding PCO Name ($PCO_Id), except that it is prefixed by either "xIOR" (Form 1), or by "xRFILE" (Form 2), or by "xNSERV" (Form 3), but not by "PCO".

NOTE: <Instance_Id> does not necessary match corresponding value included in PCO name. This is because although each PCO corresponds to one instance of CORBA object within one Test Case, there may be many Test Cases inside ATS, each using the same PCO for its own purposes (and for representing different instances of CORBA objects).

We later refer to this "modified" <Instance_Id> as <TC_Instance_Id>. It may be recommended that persistent pools of <TC_Instance_Id> values are assigned to each Test Case for each PCO representing CORBA object.

(b) Parameter Type ($TS_ParType) must be either "IA5String" (Forms 1 and 2), or "GW_string_array" (Form 3). For definition of "GW_string_array" see section 9.

(c) Only one of three alternative forms shall be used per one Server object.

Here is an example:

```
// IDL: Declaration of interface “IntC”:
module ModA {
        module ModB {
interface IntC {
                ...
        };
    };
};

// TTCN.MP: Declaring location of “ModA.ModB.IntC”, Server object #1:

$TS_ParDcl

    $TS_ParId xIOR1_iModA_iModB_iIntC
    $TS_ParType IA5String
    $PICS_PIXITref /* */
    $Comment /* IOR of the CORBA object to be registered */
$End_TS_ParDcl

// TTCN.MP: Declaring location of “ModA.ModB.IntC”, Server object #2:

$TS_ParDcl

    $TS_ParId xRFILE2_iModA_iModB_iIntC
    $TS_ParType IA5String
    $PICS_PIXITref /* Location of the file containing IOR */
    $Comment /* */
$End_TS_ParDcl
// TTCN.MP: Declaring location of “ModA.ModB.IntC”, Server object #3:

$TS_ParDcl

    $TS_ParId xNSERV3_iModA_iModB_iIntC
    $TS_ParType GW_string_array
    $PICS_PIXITref /* */
    $Comment /* Location of CORBA Object in the Naming Service */
$End_TS_ParDcl
```

2.2.3. Declaring Registration PDU

To associate Server object with its PCO, ATS must issue a PDU through the PCO that is to be associated with CORBA object. Registartion PDU is declared only once for all PCOs that are going to be associated with Server objects. Exactly three forms of Registration PDU are defined below, and ATS shall use the most suitable one while registeging Server object. Semantics of the forms is the same as described in 2.2.2. Note that choice of form of Registration PDU must match corresponding PIXIT parameter declaration.

```
// TTCN.MP : Declaring Registration PDU (Form 1):

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pSREG_IOR
    $PCO_Type CORBA_PCO
    $Comment /* Registration PDU: Form 1, using IOR */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId IOR
        $PDU_FieldType IA5String
```

```
        $Comment /* IOR of the CORBA object to be registered */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef

// TTCN.MP : Declaring Registration PDU (Form 2):

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pSREG_RFILE
    $PCO_Type CORBA_PCO
    $Comment /* Registration PDU: Form 2, using IOR location */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId RFILE
        $PDU_FieldType IA5String
        $Comment /* Location of the file containing IOR */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef

// TTCN.MP : Declaring Registration PDU (Form 3):

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pSREG_NSERV
    $PCO_Type CORBA_PCO
    $Comment /* Registration PDU: Form 3, using Naming Service */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId NSERV
        $PDU_FieldType GW_string_array
        $Comment /* Location of CORBA Object in the Naming Service */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef
```

2.2.4. Declaring Constraint on Registration PDU

In addition to PDU declaration, a corresponding constraint must be defined. Parameterised constraint declaration is used for this purpose. Exactly three forms of constraints are defined below, and ATS shall use the most suitable one while registeging Server object. Semantics of the forms is the same as described in 2.2.2. Choice of constraint form must match corresponding PIXIT parameter declaration.

```
// TTCN.MP : Constraint on Registration PDU (Form 1):

$Begin_TTCN_PDU_Constraint

    $ConsId cSREG_IOR (reference : IA5String)
    $PDU_Id pSREG_IOR
    $DerivPath
    $Comment /* Registration Constraint: Form 1, using IOR */
```

```
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId IOR
        $ConsValue reference
        $Comment /* IOR of the CORBA object to be registered */
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint

// TTCN.MP : Constraint on Registration PDU (Form 2):

$Begin_TTCN_PDU_Constraint

    $ConsId cSREG_RFILE (reffile : IA5String)
    $PDU_Id pSREG_RFILE
    $DerivPath
    $Comment /* Registration Constraint: Form 2, using IOR location */
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId RFILE
        $ConsValue reffile
        $Comment /* Location of the file containing IOR */
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint

// TTCN.MP : Constraint on Registration PDU (Form 3):

$Begin_TTCN_PDU_Constraint

    $ConsId cSREG_NSERV (nslocation : GW_string_array)
    $PDU_Id pSREG_NSERV
    $DerivPath
    $Comment /* Registration Constraint: Form 3, using Naming Service */
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId NSERV
        $ConsValue nslocation
        $Comment /* Location of CORBA Object in the Naming Service */
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint
```

2.2.5. Issuing Registration PDU

Now everything is ready for issuing a Registration PDU in Test Case. This is done in a manner illustrated by an example:

```
// TTCN.MP : Issuing Registration PDU for Server objects derived from IDL // interface “ModA.ModB.IntC”:
```

. . .
$BehaviourLine

$LabelId
$Line [0] PCO1_iModA_iModB_iIntC ! pSREG_IOR
$Cref cSREG_IOR (xIOR3_iModA_iModB_iIntC)
$Comment /* Notice that parameter name does not strictly match
PCO name - see note in section 2.2.2.(a) for explanation */
$End_BehaviourLine
$BehaviourLine

$LabelId

$Line [0] PCO2_iModA_iModB_iIntC ! pSREG_NSERV $Cref cSREG_NSERV
(xNSERV5_iModA_iModB_iIntC)

$End_BehaviourLine
. . .

Upon receipt of this PDU, Gateway will associate already running Server object with the PCO through which Registration PDU has been sent. Since then, all operations going through this PCO will be relayed to the registered CORBA object.

Gateway will respond with a standard Gateway Exception PDU indicating how successful an operation was. Format and semantics of Gateway Exception PDU will be covered in a separate section (note that CALL_ID in this case is always set to 0).

2.3. Registering Client Objects

Clients need to know location of the server that is going to handle their requests. Thus, registering Clients will normally go through the following steps:

(a) ATS sends a Registration PDU to the Gateway. Upon receipt of this PDU, Gateway will create a universal CORBA servant capable of handling requests from Clients expressed in arbitrary form (by using DSI interface). Gateway will also associate newly created Servant with PCO through which Registration PDU has been sent. Gateway will either create file containing IOR of the servant or put it into Naming Service to advertise its presence to the Client.
(b) Client may regularly check a well-defined location in order to detect whether IOR file has been created. As soon as IOR file appears, Client may may connect itself to Universal Servant. The same is possible for Naming Service. Alternatively, ATS may express a manual control over SUT (see section 10), i.e. as soon as all relevant Universal Servants are created, ATS may ask operator to connect Clients to them to manually by issuing a corresponding Manual PDU.
(c) Since then, Universal Servant will convert all requests coming from Client into TTCN PDUs following well-defined rules described in consequent sections. It will then further relay them to ATS. Upon receipt of such PDU, ATS will handle it and respond back to Client by some other PDU via PCO associated with Universal Servant. In other words, Test Suite introduces functionality to the Universal Servant. Thus, Universal Servant together with ATS emulate behaviour of a real servant.

2.3.1. Declaring PCOs for Client Objects

PCOs for Client objects must be declared exactly as defined in section 2.1. IDL description of a corresponding Universal Servant is taken as a base for constructing PCO Name.

2.3.2. Declaring Location of Universal Servant

Parameter declaration follows the same rules as defined in 2.2.2., except that only Forms 2 and 3 can be used.

2.3.3. Declaring Registration PDU for Client Objects

Declaration of Registration PDU follows the same rules as defined in 2.2.3., except that PDU Name ($PDU_Id) is prefixed by “pCREG”, not by “pSREG”, and only Forms 2 and 3 can be used.

2.3.4. Declaring Constraint on Registration PDU

Declaration of constraint on Registration PDU follows the same rules as defined in 2.2.4., except that Constraint Name ($ConsId) is prefixed by “cCREG”, not by “cSREG”; PDU Name ($PDU_Id) is prefixed by “pCREG”, not by “pSREG”; only Forms 2 and 3 can be used.

2.3.5. Issuing Registration PDU

Now everything is ready for issuing a Registration PDU in Test Case. This is done in a manner illustrated by an example:

```
// TTCN.MP : Issuing Registration PDU for Client object(s) to be connected // to the Universal Servant derived from IDL interface “ModA.ModB.IntC”:


...
$BehaviourLine
        $LabelId
        $Line [0] PCO1_iModA_iModB_iIntC ! pCREG_RFILE $Cref cCREG_RFILE
        (xRFILE7_iModA_iModB_iIntC)
        $Comment /* Notice that parameter name does not strictly match
PCO name - see note in section 2.2.2.(a) for explanation */
$End_BehaviourLine
...
```

All the preliminary declarations necessary to launch the example above are listed below:

```
// IDL: Declaration of a Universal Servant interface “ModA.ModB.IntC” that // handles requests of Client(s):

module ModA {
        module ModB {
interface IntC {
                ...
          };
     };
};

// TTCN.MP: Declaring PCO for Universal Servant “ModA.ModB.IntC”:

$PCO_Dcl

    $PCO_Id PCO1_iModA_iModB_iIntC
    $PCO_TypeId CORBA_PCO
    $PCO_Role LT
    $Comment /* First instance of “IntC” */
$End_PCO_Dcl

// TTCN.MP: Declaring location of Universal Servant IOR file:


$TS_ParDcl
    $TS_ParId xRFILE7_iModA_iModB_iIntC
    $TS_ParType IA5String
    $PICS_PIXITref /* */
    $Comment /* */
$End_TS_ParDcl
```

```
// TTCN.MP : Declaring Registration PDU for all Universal Servants:

$Begin_TTCN_PDU_TypeDef
    $PDU_Id pCREG_RFILE
    $PCO_Type CORBA_PCO
    $Comment /* Declaration of Registration PDU */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId RFILE
        $PDU_FieldType IA5String
        $Comment /* File with IOR of the Universal Servant */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef

// TTCN.MP : Constraint on Registration PDU for all Universal Servants:

$Begin_TTCN_PDU_Constraint
    $ConsId cCREG_RFILE (reffile : IA5String)
    $PDU_Id pCREG_RFILE
    $DerivPath
    $Comment /* Declaration of constraint on Registration PDU */
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId RFILE
        $ConsValue reffile
        $Comment /* IOR location of the CORBA object to be registered */
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint
```

Upon receipt of this PDU, Gateway will create a Universal Servant and save its IOR to file specified in “RFILE” field of the PDU. It will respond with a standard Gateway Exception PDU immedately if there was an error in creating Servant. Now it is expected that corresponding Client(s) will be connected to Servant provided that the IOR of the Universal Servant is already saved into file.

2.4. Discussing Alternative Registration Schemes

There have been several other proposals for registration process, namely:

(a) launching Gateway with appropriate options;
(b) using definitions of TTCN operations inside ATS.

However, current design has been chosen finally due to the following reasons:

First approach was not approved since Gateway is viewed as a fixed system constantly running in CORBA Universe. Lauching a Gateway every time ATS is to be executed is viewed as something very undesirable.

Second approach has been declined for the sake of leaving Tester part of the Test System as generic as possible. Overloading Tester with new operations and introducing unnecessary complexity to it has not been prefered either.

Present design leaves all details of handling registration process to the Gateway, what seem to be the most natural solution. Moreover, Test Case reflects stages of registration in a clear and understandable manner.

3. Deregistering Server and Client Objects

Test Case shall not worry about deregistration of association between PCOs and Servers/Clients. This will be done automatically by the Gateway upon completion of a Test Case. All proxies for Servers will be removed, as well as all Universal Servants created for Clients.

4. Operation Calls to Server Objects

4.1. Declaring Call PDU for Operation Invocation

In order to invoke an operation, ATS must send a Call PDU to Server object through the PCO associated with it. TTCN PDU declaration must adhere to the following format:

(a) Call PDU is declared only once for all invocations of one explicit operation belonging to one explicit interface.
(b) PDU Name follows the same rules as defined in section 2.1., except that:

[1] It is prefixed by “pCALL”, not by “PCO”;
[2] <Instance_Id> construct is omitted;
[3] PDU Name is posfixed by the following construct:

[”_i”] [<Op_Name>]

Here <Op_Name> is the name of the operation as it is defined in IDL. All underscore characters in the <Op_Name> must be doubled.

(c) PCO Type must be declared as “CORBA_PCO”.
(d) The first field of the PDU must be called CALL_ID and be of type INTEGER.
(e) The second field of the PDU must be called CONTEXT and be of type GW_string_array, in case “context” declaration is present in the definition of the corresponding operation. For definition of type “GW_string_array” see section 9. This field is omitted in case no “context” is declared for this operation. If “context” is declared, then CONTEXT field contains array with even amount of string objects. String objects with even indeces in this array correspond to property names and string objects with odd indeces correspond to property values. Notice that during invocation it is not prohibited to omit properties even if they are listed in “context” clause of operation. However, those properties that are not mentioned there will not be passed neither to Server from ATS, nor from Client to ATS. Even if all properties were omitted, CONTEXT field shall be present anyway, and it shall contain blank array of string objects.
(f) Consequent fields describe in and inout parameters in the same order as they are defined in IDL. Out parameters and return value of the operation are omitted in PDU declaration. Explicit format of these fields is described in section 7.

Here is an example:

```
// IDL: Declaration of Server operation “ModA.ModB.IntC.operD”:
module ModA {
        module ModB {
interface IntC {
                                                ... operD (...);
                . . . . . . . . . . . . . . . .
          };
     };
};
```

// TTCN.MP : Declaring Call PDU for invocation of operation “operD” of // interface “ModA.ModB.IntC”:

```
$Begin_TTCN_PDU_TypeDef
    $PDU_Id pCALL_iModA_iModB_iIntC_ioperD
    $PCO_Type CORBA_PCO
    $Comment /* Declaration of Call PDU */
    $PDU_FieldDcls

    $PDU_FieldDcl
        $PDU_FieldId CALL_ID
        $PDU_FieldType INTEGER
        $Comment /* Helps to distinguish between consequent invocations of
    operation “operD” of instance #1 of interface “ModA.ModB.IntC” */
    $End_PDU_FieldDcl

       . . . . . .

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef
```

4.2. Declaring Constraint on Call PDU

Constraints on Call PDU must be declared as follows:

(a) New constraint must be declared for each invocation of an operation.
(b) Constraint Name follows the same rules as defined for PDU Name in 4.1., except that it is prefixed not by “pCALL”, but by the following construct:

    [”c”] [<Invocation_Id>] [”CALL”] [<TC_Instance_Id>]

    Here <Invocation_Id> is a decimal number enumerating consequent invocations of one explicit method within one Test Case. Invocations are enumerated in ascending order starting from 1. <TC_Instance_Id> has the same semantics as described in section 2.2.2.(a).

(c) PDU Name is a name of the corresponding PDU (see section 4.1.).
(d) Value of CALL_ID field must be unique for each new invocation of any Server operation within a single Test Case.
(e) CONTEXT field (if any) contains operation context as defined in OMG CORBA specification.
(f) All other features of constraint definition (including parametrization) are left at the will of ATS designer.

Here is an example:

```
// TTCN.MP : Constraint on Call PDU for invocation #3 of operation “operD” // of instance #5 of interface
“ModA.ModB.IntC”:

$Begin_TTCN_PDU_Constraint
    $ConsId c3CALL5_iModA_iModB_iIntC_ioperD
    $PDU_Id pCALL_iModA_iModB_iIntC_ioperD
    $DerivPath
    $Comment /* Declaration of constraint on Call PDU */
    $PDU_FieldValues

    $PDU_FieldValue
        $PDU_FieldId CALL_ID
        $ConsValue 27
        $Comment /* ID of an operation invocation within a Test Case */
    $End_PDU_FieldValue
```

. . . . . .

$End_PDU_FieldValues
$Comment /* */
$End_TTCN_PDU_Constraint

4.3. Issuing Call PDU within Test Case

Now everything is ready for issuing a Call PDU in Test Case. This is done in a manner illustrated by an example:

```
// TTCN.MP : Issuing 3-rd Call PDU for Server object #5:

. . .
$BehaviourLine
        $LabelId
        $Line [0] PCO1_iModA_iModB_iIntC ! pCALL_iModA_iModB_iIntC_ioperD
        $Cref c3CALL5_iModA_iModB_iIntC_ioperD
$End_BehaviourLine
. . .
```

Gateway will create a Dynamic Invocation Request (DIR) and invoke remote CORBA operation. It will wait until: (a) operation returns or (b) exception is thrown or © Test Case is finished. In first two cases Gateway will respond to Tester either with Reply PDU, or with Exception PDU. Format of Exception PDU is covered in section 9. Gateway may also respond with Fatal Gateway Exception PDU in case of error caused by testing environment rather than by SUT.

4.4. Declaring Reply PDU for Operation Response

Once operation is invoked by issuing Call PDU, it will normally respond back with Reply PDU. Its declaration follows the same rules as defined in section 4.1., except that it is prefixed by “pREPLY”, not by “pCALL”.

CONTEXT field is always omitted and fields after CALL_ID describe out and inout parameters in the same order as they are defined in IDL. Explicit format of these fields is described in section 7. Field right after CALL_ID describes return value of the operation and is always called “RET_value”. If return value is “void”, then this field is omitted. “In” parameters are omitted in PDU declaration.

4.5. Declaring Constraint on Reply PDU

Constraint declaration is the same as defined in section 4.2., except that it is prefixed e.g. by “c3REPLY”, not by “c3CALL”, and refers to Reply PDU, not to Call PDU. Gateway guarrantees that CALL_ID field will match the value contained in the corresponding Call PDU.

4.6. Waiting for Reply PDU within Test Case

This is done as illustrated by an example:

```
// TTCN.MP : Waiting for 3-rd Reply PDU from Server object #5:
```

. . .
$BehaviourLine

$LabelId

$Line [1] PCO1_iModA_iModB_iIntC ? pREPLY_iModA_iModB_iIntC_ioperD

$Cref c3REPLY5_iModA_iModB_iIntC_ioperD

$End_BehaviourLine
. . .

NOTE: Gateway may also respond with Fatal Gateway Exception PDU in case of error caused by testing environment rather than by SUT.

5. Operation Calls from Client Objects

Operation calls from Client objects obey exactly the same rules as defined in section 4, except the following:

(a) ATS must first wait for Call PDU coming from SUT and then reply with an appropriate Reply PDU.
(b) CALL_ID is assigned by the Gateway a unique value for each new operation call coming from Client. However, it may occasionally collide with CALL_ID values assigned by ATS for Server operation calls. No conflict problems arise from this fact, however.

CALL_ID field must be stored by ATS as a variable and used in corresponding Reply PDU.

(c) CALL_ID field of constraint on Call PDU must contain "?", i.e. "any value", since ATS normally never knows excplicit value of this field generated by the Gateway. Gateway guarrantees only uniqueness of this field within a single Test Case.
(d) <Invocation_Id> parts of constraint names as defined in section 4.2. must be identical for corresponding pairs of Call and Reply PDUs.

6. Oneway Invocations

In case operation call is defined in IDL with "oneway" attribute, ATS is expected to behave as follows:

(a) If "oneway" is declared in server-side IDL of SUT, then ATS shall not expect any response from Gateway upon sending a Call PDU to SUT, except maybe Gateway or CORBA Exception. Gateway will make a best effort to deliver a call to SUT - but no guarrantee. No special care shall be taken in order to inform the Gateway about oneway nature of invocation since Gateway may obtain that kind of information from Interface Repository.
(b) If "oneway" is declared in server-side IDL of ATS (i.e. in declaration of Universal Servant), then ATS shall not respond with Reply PDU or any other PDU upon receipt of Call PDU. No special care shall be taken in order to inform the Gateway about oneway nature of invocation since Gateway may obtain that kind of information from Interface Repository.

7. Declaring Parameters in Call PDU and Reply PDU

First thing to remember about parameter types is that all of them are defined using ASN.1. This is because native TTCN types provide poor means for mapping of IDL to TTCN. On the other hand, definitions of ASN.1 structured types do not allow embedding of native TTCN types.

7.1. Conventions on PDU Field Names

Each PDU Field Name ($PDU_FieldId) that corresponds to a specific operation parameter must be constructed as follows:

(a) It is prefixed by “PAR_” substring.
(b) It is postfixed by parameter name as defined in IDL interface.

E.g. for IDL parameter “in string hostName” the following line will be present in PDU declaration:

$PDU_FieldId PAR_hostName

7.2. Simple Types

The following simple IDL types must be defined in ATS exactly in the manner represented below. Parameters of simple types defined inside Call PDU or Reply PDU must later refer to these definitions while declaring parameter type.

```
// TTCN.MP : Declaration of simple IDL types

$ASN1_TypeDefs

. . .
$Begin_ASN1_TypeDef
    $ASN1_TypeId IDL_short
    $ASN1_TypeDefinition INTEGER (-32768..32767)
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL short type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

$ASN1_TypeId IDL_long
    $ASN1_TypeDefinition INTEGER (-2147483648..2147483647)
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL long type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

$ASN1_TypeId IDL_long_long
    $ASN1_TypeDefinition INTEGER (-9223372036854775808..9223372036854775807)

$End_ASN1_TypeDefinition
    $Comment /* Definition of IDL long long type */

$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_unsigned_short
    $ASN1_TypeDefinition INTEGER (0..65535)
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL unsigned short type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_unsigned_long
    $ASN1_TypeDefinition INTEGER (0..4294967295)
    $End_ASN1_TypeDefinition
```

```
    $Comment /* Definition of IDL unsigned long type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

$ASN1_TypeId IDL_unsigned_long_long
    $ASN1_TypeDefinition INTEGER (0..18446744073709551615)

$End_ASN1_TypeDefinition
    $Comment /* Definition of IDL unsigned long long type */

$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_char
    $ASN1_TypeDefinition IA5String(SIZE(1))
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL char type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_wchar
    $ASN1_TypeDefinition BMPString(SIZE(1))
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL wchar type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_string
    $ASN1_TypeDefinition IA5String
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL string type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_wstring
    $ASN1_TypeDefinition BMPString
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL wstring type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_octet
    $ASN1_TypeDefinition OCTET STRING (SIZE(1))
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL octet type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_boolean
    $ASN1_TypeDefinition BOOLEAN
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL boolean type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_float
    $ASN1_TypeDefinition REAL
    $End_ASN1_TypeDefinition
```

```
    $Comment /* Definition of IDL float type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_double
    $ASN1_TypeDefinition REAL
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL double type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_long_double
    $ASN1_TypeDefinition REAL
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL long double type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_fixed
    $ASN1_TypeDefinition REAL
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL fixed type */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId IDL_object
    $ASN1_TypeDefinition INTEGER
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL object reference type */
$End_ASN1_TypeDef

...
$End_ASN1_TypeDefs
```

Notice that is also possible to send IDL octet from Tester to SUT in the form of INTEGER (-128..255), although conventional way of doing this (i.e. using OCTETSTRING) is still preferred. IDL octet always arrives from SUT to Tester in the form of OCTETSTRING, as defined in ASN.1 type IDL_octet.

7.2.1. Bounded String Type

It is possible in IDL to define a bounded string type, i.e. string type with maximum size specified. If necessary, ATS designer must specify an appropriate type in Test Suite in a manner illustrated by an example (note that Type Name in TTCN is prefixed by “TYPE_” indicating mapping from IDL type definition):

```
// IDL : Definition of a bounded string type:

interface MobileIP {

    typedef string<100> packet;

    ...
};

// TTCN.MP : Definition of a bounded string type:

$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_ipacket
    $ASN1_TypeDefinition IA5String(SIZE(0..100))
```

```
    $End_ASN1_TypeDefinition
    $Comment /* Definition of IDL bounded string type */
$End_ASN1_TypeDef
```

7.3. Type Declarations

Mapping of IDL type declaration (“typedef”) to TTCN is illustrated by an example from section 7.2.1. Note that in order to avoid name scoping problems with type declarations, definition of “packet” type is prefixed by name scope for the location where this type is defined. This obeys the rules similar to those defined in section 2.1.

There is another thing worth to remember about type declarations in IDL. If there is no explicit type declaration in IDL file for a complex type like enum, struct, union, sequence, array, bounded string etc., ATS designer must make any reasonable assumption about how this definition could have looked like and declare it in a manner as described in an example from section 7.2.1.

7.4. Enumerated Type

Mapping of enumerated types from IDL to TTCN is decribed by an example below. Again, take a notice of measures taken to avoid name scoping conflicts (note also that all enumerated values are preceded by a name scope to avoid name scope conflicts):

```
// IDL : Definition of enumerated type:

interface MobileIP {

    enum colors { red, green, blue };

      . . .
};

// TTCN.MP : Definition of enumerated type:
$Begin_ASN1_TypeDef
        $ASN1_TypeId TYPE_iMobileIP_icolors
        $ASN1_TypeDefinition ENUMERATED { enum_iMobileIP_ired(0),
enum_iMobileIP_igreen(1), enum_iMobileIP_iblue(2) }
        $End_ASN1_TypeDefinition
        $Comment /* Definition of IDL enumerated type */
$End_ASN1_TypeDef
```

Notice that is also possible to send IDL enumerated value from Tester to SUT in the form of INTEGER (0..4294967295), although conventional way of doing this (i.e. using ENUMERATED) is still preferred, as long as it promotes higher level of ATS readability.

7.5. Sequence Type

7.5.1. Unbounded Sequences

Mapping of unbounded sequence types from IDL to TTCN is illustrated by an example below:

```
// IDL : Definition of unbounded sequence type:

interface MobileIP {
```

```
    typedef sequence<octet> packet;
        . . .
};

// TTCN.MP : Definition of unbounded sequence type:
$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_ipacket
    $ASN1_TypeDefinition SEQUENCE OF IDL_octet
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

7.5.2. Bounded Sequences

Mapping of bounded sequence types from IDL to TTCN is illustrated by an example below:

```
// IDL : Definition of bounded sequence type:
interface MobileIP {
    typedef sequence<octet,100> packet;
        . . .
};

// TTCN.MP : Definition of bounded sequence type:
$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_ipacket
    $ASN1_TypeDefinition SEQUENCE SIZE(0..100) OF IDL_octet
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

7.5.3. Nested Sequences

It is possible in IDL that a sequence type may be used as the type parameter for another sequence type. This is possible in TTCN also:

```
// IDL : Definition of nested sequence type:
interface MobileIP {
    typedef sequence<sequence<long> > registry;
        . . .
};

// TTCN.MP : Definition of nested sequence type:
$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_iregistry
    $ASN1_TypeDefinition SEQUENCE OF SEQUENCE OF IDL_long
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

7.5.4. Sequences with Zero Length

Notice that IDL does not prohibit sequences to be of zero length at run time. If this is the case, corresponding definition of constraint shall contain brackets "{" and "}" with blank content inside brackets, rather than omit the whole structure at all.

7.6. Array Type

Mapping of array types from IDL to TTCN is illustrated by an example below:

```
// IDL : Definition of array type:
interface MobileIP {
    typedef long table[10][5];
      . . .
};

// TTCN.MP : Definition of array type:
$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_itable
    $ASN1_TypeDefinition SEQUENCE SIZE(10) OF SEQUENCE SIZE(5) OF IDL_long
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

7.7. Structured Type

IDL structured type maps to TTCN as illustrated by an example (note that all field names are preceded by underscore character "_" to avoid conflicts with TTCN and ASN.1 reserved words):

```
// IDL : Definition of structured type:
interface MobileIP {
    struct human
      {
    short age;
    string name;
      };
      . . .
};

// TTCN.MP : Definition of structured type:
$Begin_ASN1_TypeDef
    $ASN1_TypeId TYPE_iMobileIP_ihuman
    $ASN1_TypeDefinition SEQUENCE { seq_age IDL_short, seq_name IDL_string }
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

7.8. Union Type

IDL union type maps to TTCN as illustrated by an example (note that all field names are preceded by underscore character "_" to avoid conflicts with TTCN and ASN.1 reserved words):

```
// IDL : Definition of union type:

interface MobileIP {

    union register switch (char)

      {
    case 'a': case 'b': short AX;

    case 'c': long EAX;

    default:  octet AL;

      };
      ...
};

// TTCN.MP : Definition of union type:

$Begin_ASN1_TypeDef

    $ASN1_TypeId UNION_iMobileIP_iregister
    $ASN1_TypeDefinition CHOICE
      {
       ch_AX  [0] IDL_short,
       ch_EAX [1] IDL_long,
       ch_AL  [2] IDL_octet
      }
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef

    $ASN1_TypeId TYPE_iMobileIP_iregister
    $ASN1_TypeDefinition SEQUENCE
      {
       seq_discriminator IDL_char,
       seq_value UNION_iMobileIP_iregister
      }
    $End_ASN1_TypeDefinition
    $Comment /* */
$End_ASN1_TypeDef
```

Note that in order to access "default" section of "union", "seq_discriminator" shall contain any value not mentioned in "case" sections.

7.9. ANY Type

Present specification does not provide any support for IDL any type. Nowadays any type is viewed as a gap in a system specification, necessary only for low-level system design.

If ANY type is present in declaration of CORBA operation under test, it is recommended to implement a Decorator (as it is defined in design patterns language) which will contain one of more operations, each having ANY replaced with concrete IDL type, as needed by one or more corresponding test steps. In this way, decorator set of operations will provide equivalence transformation of operation in question into acceptable form. Implemented Decorator shall be viewed as an integral part of SUT intended to facilitate SUT testing.

8. IDL Attribute Declaration

Declaration of IDL attribute is equivalent to get/set pair of IDL operations as described in OMG CORBA specification. Hence, no additional rules shall be introduced with concern to attributes and their handling.

We draw reader's attention to the fact that OMG IDL specification implicitly assumes that the name of the parameter passed as agrument to set() operation call shall be equivalent to the first letter in the name of the corresponding attribute. Making this assumption explicit, we require from set() operation call to assign name of the argument exactly as described above.

9. Exception Handling

In order to be able to throw an exception or to handle an exception, ATS must declare Exception PDU. It is declared only once for all exceptions thrown and for all exceptions caught and is used in the ATS later multiple times with appropriate constraints.

ATS shall be prepared to catch exceptions declared after "raises" keyword. It shall be prepared also to catch CORBA System Exceptions since any operation may throw them without explicit declaration. Gateway may throw its own exceptions as well. In case an exception is thrown by a Server during operation invocation, ATS shall not expect any other response from this operation, nor shall it respond to a Client in any other way in case it decides to throw its own exception as a response to a Client request.

9.1. Handling CORBA System Exceptions

Exactly the following definitions must be present in ATS in order to handle CORBA System Exceptions:

```
// IDL: Definition of CORBA standard exception body:
module CORBA {
        interface SystemException {
    enum completion_status
    { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE };
    #define ex_body { unsigned long minor; completion_status completed; }
      . . .
};

// TTCN.MP : Definition of "completion_status" enumerated type:
$Begin_ASN1_TypeDef
        $ASN1_TypeId TYPE_iCORBA_iSystemException_icompletion__status
        $ASN1_TypeDefinition ENUMERATED {
enum_iCORBA_iSystemException_iCOMPLETED__YES(0),
        enum_iCORBA_iSystemException_iCOMPLETED__NO(1),
enum_iCORBA_iSystemException_iCOMPLETED__MAYBE(2) }
        $End_ASN1_TypeDefinition
        $Comment /* Definition of "completion_status" enumerated type */
$End_ASN1_TypeDef
// TTCN.MP : Definition of "ex_body" structured type:
$Begin_ASN1_TypeDef
        $ASN1_TypeId TYPE_iCORBA_iSystemException_iex__body
        $ASN1_TypeDefinition SEQUENCE {
seq_minor IDL_unsigned_long,
seq_completed TYPE_iCORBA_iSystemException_icompletion__status }
        $End_ASN1_TypeDefinition
```

```
        $Comment /* Definition of “ex_body” structured type */
$End_ASN1_TypeDef
```

9.2. Gateway-Specific Type Definitions

Exactly the following Gateway-specific definitions must be present in ATS:

```
$Begin_ASN1_TypeDef

    $ASN1_TypeId GW_string_array
    $ASN1_TypeDefinition SEQUENCE OF IA5string
    $End_ASN1_TypeDefinition
    $Comment /* Definition of Gateway-specific array of IDL strings */
$End_ASN1_TypeDef

$Begin_ASN1_TypeDef
        $ASN1_TypeId GW_void_exc_body
        $ASN1_TypeDefinition SEQUENCE { seq_error_code IDL_long }
        $End_ASN1_TypeDefinition
        $Comment /* Definition of “void” exception body */
$End_ASN1_TypeDef
```

9.3. Declaring Exception PDU

Exactly the following definition for Exception PDU must be present in ATS:

```
$Begin_TTCN_PDU_TypeDef

    $PDU_Id pRAISE
    $PCO_Type CORBA_PCO
    $Comment /* Declaration of Exception PDU. */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId CALL_ID
        $PDU_FieldType INTEGER
        $Comment /* ID of the operation call that has thrown an exception. */
    $End_PDU_FieldDcl

    $PDU_FieldDcl

    $PDU_FieldId EXC_SCOPE
    $PDU_FieldType IDL_string_array
    $Comment /* Name scope where exception has been initially defined. */
    $End_PDU_FieldDcl

    $PDU_FieldDcl

    $PDU_FieldId EXC_NAME
    $PDU_FieldType IA5String
    $Comment /* Exception name. */
    $End_PDU_FieldDcl

    $PDU_FieldDcl

    $PDU_FieldId EXC_BODY
    $PDU_FieldType PDU
```

```
    $Comment /* Exception body. */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef
```

CALL_ID identifies operation that caused exception. In case exception is thrown by the Gateway or it is impossible to detect which operation thrown an exception, CALL_ID is set to 0.

EXC_SCOPE contains sequence of modules and interfaces forming a name scope for an exception declaration. NOTE: Gateway assumes that all standard CORBA exceptions are defined within an interface “CORBA.SystemException”. Gateway may throw its own exceptions, and these are defined within an interface called “GatewayException”.

EXC_NAME contains name of an exception thrown.

EXC_BODY is a body of exception.

9.4. Declaring Body of Exception PDU

Declaration of EXC_BODY is illustrated for CORBA System Exceptions:

```
// IDL: Definition of CORBA System Exceptions:

module CORBA {

    interface SystemException {

          . . .
    exception UNKNOWN { unsigned long minor; completion_status completed; }; exception BAD_PARAM {
    unsigned long minor; completion_status completed; };

       . . .
   };
};

// TTCN.MP : Declaring body for CORBA System Exception PDU:

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pRAISE_BODY_iCORBA_iSystemException
    $PCO_Type CORBA_PCO
    $Comment /* Declaration of Exception PDU body. */
    $PDU_FieldDcls

$PDU_FieldDcl
                                        $PDU_FieldId EXC_BODY
                                        $PDU_FieldType TYPE_iCORBA_iSystemException_iex__body
                                        $Comment /* Exception body. */
$End_PDU_FieldDcl
    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef

. . .
```

EXC_BODY is a body of exception as defined in a corresponding IDL definition. Exception body is always interpreted as IDL structure and is mapped to TTCN as defined in section 7.6. If exception has no body, it is assigned a default body of type GW_void_exc_body as defined in section 9.2. Field “seq_error_code” always contains 0. Structure of EXC_BODY may vary for different exceptions.

9.5. Declaring Constraint on Exception PDU

Assume that ATS is eager to catch all CORBA System Exceptions that may be thrown by any of currently invoked operations. In this case it should wait for receipt of Exception PDU constrained by the following construct:

```
// IDL: Definition of CORBA System Exceptions:

module CORBA {

    interface SystemException {

          . . .
    exception UNKNOWN { unsigned long minor; completion_status completed; }; exception BAD_PARAM {
    unsigned long minor; completion_status completed; };

      . . .
   };
};

// TTCN.MP: Catching all CORBA System Exceptions:

$Begin_TTCN_PDU_Constraint

    $ConsId c3RAISE_iCORBA_iSystemException
    $PDU_Id pRAISE
    $DerivPath
    $Comment /* Declaration of constraint on Exception PDU */
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId CALL_ID
        $PDU_ConsValue ?
    $End_PDU_FieldValue

$PDU_FieldValue
                                                    $PDU_FieldId EXC_SCOPE
                                                    $PDU_ConsValue { “CORBA”, “SystemException” }
$End_PDU_FieldValue
    $PDU_FieldValue

    $PDU_FieldId EXC_NAME
    $PDU_ConsValue ?
    $End_PDU_FieldValue

    $PDU_FieldValue

    $PDU_FieldId EXC_BODY
    $PDU_ConsValue ?
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint
```

Here is an example of catching a user-defined exception:

```
// IDL: Definition of an exception “PacketTooBig”:

module ModA {
          module ModB {
interface IntC {
```

```
                                        exception PacketTooBig { };
            ...
        };
    };
};
```

// TTCN.MP: Catching an exception “PacketTooBig”:

$Begin_TTCN_PDU_Constraint

$ConsId c3RAISE_iModA_iModB_iIntC_iPacketTooBig

$PDU_Id pRAISE
$DerivPath
$Comment /* Declaration of constraint on Exception PDU. */
$PDU_FieldValues

$PDU_FieldValue

$PDU_FieldId CALL_ID
$PDU_ConsValue ?
$End_PDU_FieldValue

$PDU_FieldValue

$PDU_FieldId EXC_SCOPE

$PDU_ConsValue { “ModA”, “ModB”, “IntC” }

$End_PDU_FieldValue

$PDU_FieldValue

$PDU_FieldId EXC_NAME
$PDU_ConsValue “PacketTooBig”
$End_PDU_FieldValue

$PDU_FieldValue

$PDU_FieldId EXC_BODY
$ConsValue ?
$End_PDU_FieldValue

$End_PDU_FieldValues
$Comment /* */
$End_TTCN_PDU_Constraint

9.6. Exceptions Defined by Gateway

Currently, the following exceptions are defined by the Gateway. Their description is given in IDL-style:

```
module GatewayException {
    // Exceptions defined here indicate successful events:
    interface Recoverable {
    // This exception is generally thrown for successful events: exception General { };

      };
    // Exceptions defined here indicate fatal events:
    interface Fatal {
```

```
    // This exception is generally thrown for fatal events: exception General { };
    };
};
```

9.7. Discussion on Specified Exception Handling

Alternative approaches to handling CORBA exceptions in ATS are, of course, possible. However, present design have been chosen finally due to following reasons:

(a) Exception PDU contains information sufficient to detect which exception has been thrown and in which name scope it is defined.
(b) It contains information sufficient to detect which operation has thrown an exception.
(c) Body of exception is present as a PDU field.
(d) Groups of exceptions can be caught easily within a Test Case by defining an appropriate constraint, as illustated by an example above.

10. Manual Control over SUT

It may sometimes be needed for ATS to tell operator to manually perform some actions, e.g. invoke Client requests of Universal Servant operations. This is done by IMPLICITly SENDing Manual PDU to the Gateway. Format of Manual PDU is as follows:

```
// TTCN.MP : Declaring Manual PDU:

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pMANUAL
    $PCO_Type CORBA_PCO
    $Comment /* Declaration of Manual PDU */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId MESSAGE
        $PDU_FieldType IA5String
        $Comment /* Message to the operator attached */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef
```

Upon receipt of this PDU, Gateway will display a message to the operator contained in MESSAGE field directing him to perform some actions. In will also append a string: “Upon successful completion of the operation, press Y. Upon faulty completion of the operation, press N.” Once operator completes an operation and responds with an action, Gateway will respond back to ATS either with Recoverable or with Fatal Gateway exception, depending on human’s response.

ATS is expected to wait for receipt of Exception PDU coming from Gateway.

11. Inheritance and Name Resolution

IDL permits redefinition of inherited types, constants and exceptions. Since present specification requires that type, operation and exception declarations shall contain all necessary information about the scope where they have been

defined, we figure out that no difficulties shall arise in mapping of IDL resolution operator ("::") to TTCN. This operator simply defines a different name scope for operation, exception or whatever it is. ATS designer shall not also experience any problems concerned with IDL inheritance mechanism.

E.g. if resolution operator is applied to operation invocation, then Call PDU shall be send through the same PCO that would have been used in case of normal invocation, but details of defining PDU Name, constraint etc. shall consider a different name scope of an operation.

12. Handling Product Objects

Recall from section 1.1. that one distinct type of objects is Factory which dynamically creates new CORBA objects. We refer to such objects as Product objects. Currently, Product objects can be created only by SUT and can not be created by ATS.

The procedure involving creation and operation invocation of Product objects is as follows:

(a) Factory operation is invoked, and it creates Product object. Gateway will map return value of the operation representing reference to Product object to parameter of type IDL_object (see section 7.2. for definition of this type). For valid Product objects this is guarranteed to be a unique positive integer value. ATS shall save this value as a Test Case variable. This value will be later referred to as Product ID.
(b) ATS shall issue Registration PDU through the PCO assigned to Product object. Registration process is similar to described in section 2 and will be covered in next sections in detail.
(c) Now ATS operations of Product object can be invoked exactly in the same manner as described in section 4.
(d) Before completing Test Case, ATS must take care of destroying allocated Product objects by itself. Gateway will not deallocate them automatically.

12.1. Declaring PCOs for Product Objects

PCOs for Product objects must be declared exactly as defined in section 2.1. IDL description of a corresponding Product object is taken as a base for constructing PCO Name.

12.2. Declaring Product ID for Product Objects

Product ID obtained from Factory operation shall be saved to local Test Case variable. Variable declaration resembles definition of ATS PIXIT parameter covered in section 2.2.2. and is illustrated by an example:

```
// IDL: Declaration of Product interface "IntC":
module ModA {
        module ModB {
interface IntC {
                ...
        };
    };
};

// TTCN.MP: Declaring Product ID of "ModA.ModB.IntC", object #5:

$TC_VarDcl
    $TC_VarId vID5_iModA_iModB_iIntC
    $TC_VarType IDL_object
    $TC_VarValue
    $Comment /* Product ID obtained from Factory operation */
$End_TC_VarDcl
```

12.3. Declaring Registration PDU for Product Objects

Declaration of Registration PDU follows rules similar to defined in 2.2.3. Special PDU of Form 4 is introduced, and this is the only way to register Product object:

```
// TTCN.MP : Declaring Registration PDU (Form 4):

$Begin_TTCN_PDU_TypeDef

    $PDU_Id pPREG_ID
    $PCO_Type CORBA_PCO
    $Comment /* Registration PDU: Form 4, using Product ID */
    $PDU_FieldDcls

    $PDU_FieldDcl

        $PDU_FieldId ID
        $PDU_FieldType IDL_object
        $Comment /* Product ID obtained from Factory operation */
    $End_PDU_FieldDcl

    $End_PDU_FieldDcls
    $Comment /* */
$End_TTCN_PDU_TypeDef
```

12.4. Declaring Constraint on Registration PDU

Declaration of constraint on Registration PDU follows rules similar to defined in 2.2.4. Special Form 4 of constraint is introduced, and this is the only way to put constraint on Product registration:

```
// TTCN.MP : Constraint on Registration PDU (Form 4):

$Begin_TTCN_PDU_Constraint

    $ConsId cPREG_ID (productid : IDL_object)
    $PDU_Id pPREG_ID
    $DerivPath
    $Comment /* Registration Constraint: Form 4, using Product ID */
    $PDU_FieldValues

    $PDU_FieldValue

        $PDU_FieldId ID
        $ConsValue productid
        $Comment /* Product ID obtained from Factory operation */
    $End_PDU_FieldValue

    $End_PDU_FieldValues
    $Comment /* */
$End_TTCN_PDU_Constraint
```

12.5. Issuing Registration PDU

Now everything is ready for issuing a Registration PDU in Test Case. This is done in a manner illustrated by an example:

```
// TTCN.MP : Issuing Registration PDU for Product object derived from IDL // interface “ModA.ModB.IntC”:


. . .
$BehaviourLine
    $LabelId
    $Line [0] PCO1_iModA_iModB_iIntC ! pPREG_ID
    $Cref cPREG_ID (vID5_iModA_iModB_iIntC)
    $Comment /* Notice that variable name does not strictly match
PCO name - see note in section 2.2.2.(a) for explanation */
$End_BehaviourLine
. . .
```