A Test Framework for CORBA Interoperability

Mang Li[1], Arno Puder[2], Ina Schieferdecker[1]

[1]GMD FOKUS
Competence Center for Testing, Interoperability and Performance
Kaiserin-Augusta-Allee 31, D-10589 Berlin, Germany
{m.li, Schieferdecker}@fokus.gmd.de

[2]AT&T Labs
75 Willow Road
Menlo Park, CA 94025, USA
arno@att.com

Abstract

This paper presents a new approach to interoperability testing of CORBA implementations. The ORB Under Test (OUT) is embedded in the test system in such a way, that it either assumes the role of a client or a server. For a given configuration, the test system emulates the peer entity of the OUT. In this way, it is possible to generate more permutations of GIOP traffic than with other related testing approaches. The test system uses ODMG-ODL for the data schema of test cases and TTCN-3 for the behavioral specification.

Keywords

CORBA, conformance testing, interoperability, ODMG-ODL, TTCN-3

1. Introduction

Applications increasingly cross boundaries of technological, administrative and political domains. Fields like Ecommerce or Business-to-Business increase the need for cross-platform development of distributed applications. Because of the heterogeneity of these environments it is not possible to enforce or even assume one single technology. Heterogeneity exists at different levels; ranging from different network technologies, different operating systems and programming languages. Middleware platforms are proven to be an adequate means to cope with heterogeneous environments. A middleware platform spreads out like a table cloth in a heterogeneous environment, offering a well-known Application Programming Interface (API).

The Common Object Request Broker Architecture (CORBA) [3] provides a framework for the development of distributed object-oriented applications in heterogeneous environments. CORBA is published as a set of specifications that describe the behavior of a middleware platform. The specification does not prescribe implementation details.

Different vendors offering CORBA platforms can focus on different market segments and choose appropriate technical details for their implementations. As a consequence, there exist multiple CORBA implementations, ranging from commercial products to open source versions.

Deriving an implementation from a specification raises the question of conformance, i.e., if the implementation conforms to the specification. Conformance guarantees that the behavior is in accordance with the specification. Only conformant implementations guarantee interoperability and easy portability of applications between CORBA platforms. Without conformance, the very philosophy of CORBA is compromised which aims for the deployment of applications in heterogeneous environments.

This paper focuses on interoperability conformance testing of a CORBA implementation. Section 2 gives a brief overview of CORBA and explains the difference between interoperability and portability. It is argued that interoperability plays a more crucial role than portability. Section 3 first discusses several different approaches to test conformance of interoperability with their respective

drawbacks. It is followed by a proposal for a new test configuration. Section 4 then introduces a test framework based on the configuration discussed in the previous section. Section 5 finally concludes this paper with an outlook for future work.

2. CORBA and GIOP

In this section we first give a brief overview of CORBA. Our discussion focuses on the key concepts portability and interoperability of applications.

CORBA as a middleware platform provides the specification of two different kinds of interfaces which we call *horizontal* and *vertical interfaces* (see Figure 1). The horizontal interface separates the application from the middleware. The CORBA specification defines the horizontal interface amongst others through the Interface Definition Language (IDL), the Dynamic Invocation Interface (DII), Dynamic Skeleton Interface (DSI) and the Portable Object Adapter (POA).

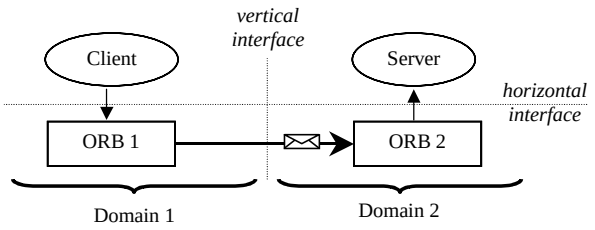

Figure 1 Vertical and horizontal interfaces

The vertical interface resides between two instances of a CORBA implementation. It is defined through the General Inter-ORB Protocol (GIOP) in the CORBA specification. The application programmer generally is unaware of the details of GIOP, while a detailed knowledge of the horizontal interface is necessary for building distributed applications. The standardization of the horizontal interface enables *portability* while the specification of the vertical interface guarantees *interoperability* of applications. Horizontal and vertical interface are not independent of each other. E.g., data types definable based on the IDL at the horizontal interface must be marshalled at the vertical interface.

In the following we argue that with respect to conformance to the CORBA specification interoperability plays a more important role than portability. As shown in Figure 1, the vertical interface decouples technological domains. The vendors of ORB1 and ORB2 can target specific markets. While a specific technological domain may require the modification of the horizontal interface, the vertical interface should not be modified in order to retain interoperability.

One example of a technological domain is embedded systems. The CORBA specification offers solutions for these environments through its MinimumCORBA specification [3]. MinimumCORBA is a true subset of the full CORBA specification in order to enable support for CORBA in resource limited environments. MinimumCORBA removes certain elements from the horizontal interface (e.g., DII and DSI) but explicitly places no restrictions on the vertical interface in order not to compromise interoperability with other CORBA platforms.

MinimumCORBA is therefore a good example where the CORBA specification itself limits portability of applications but retains full interoperability. No matter if changes to the horizontal interface compromises portability of applications, violating the interoperability is against the very core of CORBA's philosophy. In terms of conformance testing we therefore focus on interoperability.

3. Test configuration

In this section we discuss the configuration for the conformance testing of the vertical GIOP interface. First, we briefly review some related approaches and discuss their drawbacks. Then, we introduce our proposal that aims to copes with the requirements raised in the previous section.

We will use the acronym OUT to denote the *ORB Under Test*.

3.1 Related approaches

3.1.1 OUT on client and server side

In this configuration the OUT is used on both the client and the server side (i.e., ORB1 and ORB2 in Figure 1). The test suite is running on top of the OUT, invoking operations with actual parameters. The test suite makes sure that the operation reaches the server and that all parameters are transmitted correctly. This approach only guarantees the interoperability with respect to one ORB, namely the OUT. While this approach makes a statement about the possibility to build a distributed application based on the OUT, it cannot make any statements with respect to interoperability with other ORBs. In fact, the OUT could choose to implement a proprietary protocol that is not based on GIOP. This would make it impossible to interoperate with a different vendors ORB.

3.1.2 Passive testing of vertical interface

One way to assure that the protocol at the vertical interface is indeed based on GIOP is to inspect the network traffic between two instances of the OUT as the test proceeds. This approach was chosen for the conformance test suite developed by The Open Group [4]. Here a protocol analyzer sits between the client and the server OUT and

ensures that the encoding of parameters and PDUs conforms with GIOP.

While this approach can certify that the traffic between two instances of the OUT is conformant to GIOP, it does little to enforce all aspects of interoperability. The shortcomings have to do with the fact that GIOP allows certain implementation freedoms that cannot be controlled at the horizontal interface. An example for this is the endianness of the transmitted data. In GIOP, the sender can choose the endianness (little or big endian) encoding of the data to be transmitted. It is the receiver's responsibility to convert the received data into its preferred format. But GIOP does not prescribe a policy that is binding for the sender when to use which endianness. If the OUT decides to use little endian encodings for all outgoing PDUs, the observed traffic at the vertical interface will be conformant to GIOP, but the OUT will never have to demonstrate its fitness to cope with big endian encodings on the receiving side.

Since the OUT can only be controlled at its stub/skeleton interfaces, there is no way to influence when which kind of encoding will be used. There are other examples where GIOP does not prescribe specific policies that cannot be controlled at the horizontal interfaces. Examples are the padding during the alignment of data, encoding of union default members and encapsulations of embedded encodings.

3.1.3 Reference ORB implementation

To circumvent the problem mentioned in the last section one might be tempted to use a reference ORB against the OUT. But doing this yields the same problems mentioned in the previous section, as the reference ORB will choose the native endianness for the encoding of the data, just as the OUT.

Also the approach based on the request-reply paradigm (see section Section 3.1.2) may use a reference ORB. In fact, a reference implementation in certain form is always used in testing. It must behave as an ORB implementation to provide the functionality as much as the tests require. The existence of a reference ORB is in many cases only an assumption. Important is to use an adequate method and granularity so that failures can be resolved easily.

3.2 Proposal for new test configuration

The test configuration we propose in the following is to allow direct access to the GIOP interface and active control of messages exchanged with the OUT. In addition, when the test control involves the horizontal interface, its use should not restrict testing of the GIOP interface. For instance, the existence of DII and DSI should not be required to allow testing of MinimumCORBA implementations.

The test configuration is depicted in Figure 2. The OUT is highlighted by a shadowed box. Although a client-server paradigm is also used, the test components reside on different interfaces. Namely, the test client communicates with the OUT over the the vertical interface, i.e. the GIOP interface, while the test server has access to the OUT over the adaptation layer and the horizontal interface which is in this case the skeleton interface.

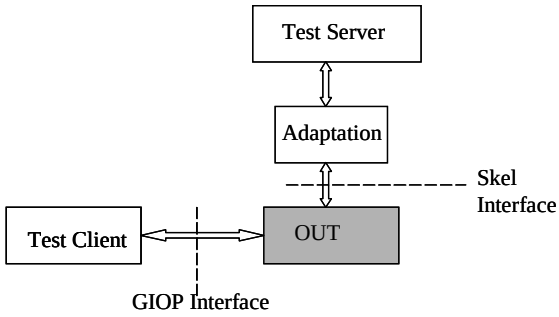

Figure 2 Test configuration

The test client is not a CORBA client because it does not use the standard stub interface. It is a test component that emulates the behavior of the counter part of the OUT. It must not be an implementation of a protocol engine. More important is that it supports the observation and assessment of the functionality of the OUT. In this sense, the test client is an active test component.

The test server uses the skeleton interface, is therefore a CORBA application. As a test component, the test server provides control of the OUT, e.g. acceptance of a request issued by the test client. It is also an active test component as it generates replies that are communicated via the OUT to the test client.

The adaptation layer is introduced for the customization of interfaces used by the test server. For example, in case the functionality of DII and DSI is used, the adaption layer may map DII and DSI to static interfaces of a MinimumCORBA ORB, while for a full CORBA ORB it can deliver the calls one-to-one to its upper and lower layers.

The activities of the test client and test server need to be coordinated, e.g. for timing of their setup, for parameterization of messages or operation invocations, or for the assignment of final test results. This can be done either manually or automatically. For the latter, a test manager can be used. It is less error-prone and more efficient.

When we leave the bi-directional GIOP for future consideration, and concentrate first on the asynchronous communication, a second test configuration beyond the one shown in Figure 2 needs to be mentioned, in which the locations of the test client and the test server are swapped. The stub interface takes the place of the skeleton interface.

With both test configurations we are able to verify the complete functionality of GIOP traffic.

In the following we continue to take the first configuration as the example but consider the testing methods so that they are applicable in both cases.

4. Specification of tests

After the test configuration is determined, its refinement in terms of test specification is considered. We propose to use high-level description in formal notations, in order to:

- allow unambiguous definition of test purposes;
- make test cases readable;
- reuse existing high-level data definitions;
- be abstract of target programming languages;
- enable high quality and efficient test development by automated code generation.

In particular, for adequate testing of GIOP traffic in terms of data transportation, we separate the focus of test specification into two major parts: test data and test case, as represented by Figure 3. It improves the scalability of the test system. Furthermore, a specification-based implementation concept is proposed. In this approach, test data covers the data structure used by the test case specification, and the data schema of the database used by the run-time of the test case implementation. In this paper we focus on the level of specification. Some ideas to the implementation is given in Section 5.

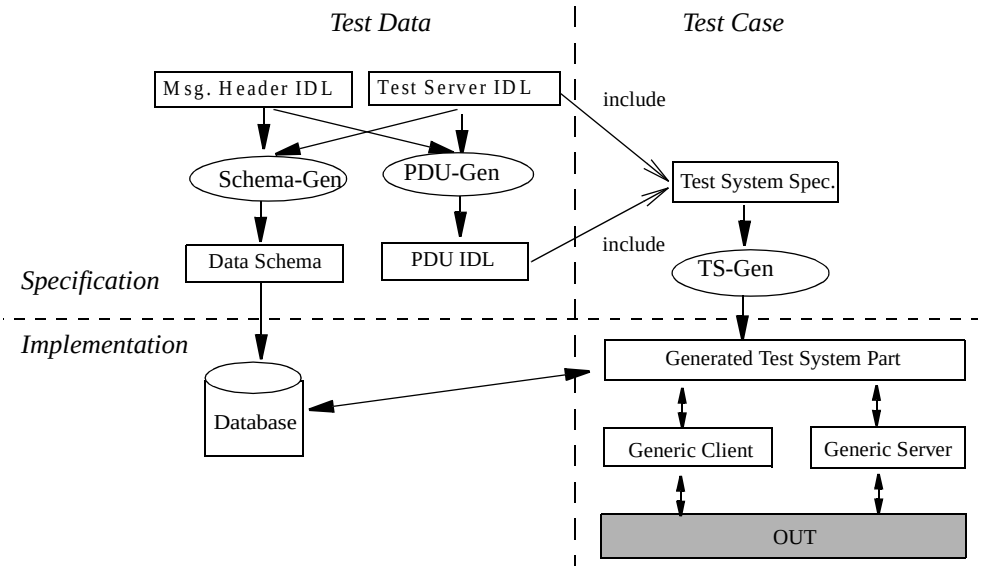

Figure 3 Test specification and implementation

In Section 4.1, we first elaborate the definition of the GIOP Protocol Data Units (PDUs) used by the test cases in CORBA IDL (abbr. IDL only), which is generated from the IDL definitions of the test server and the IDL definitions of the modules GIOP and IOP in the CORBA specification.

In Section 4.2, we proceed to the data schema that is built up from the same input, but using different transformation rules.

The specification of test cases includes the PDU data structure for access to the database derived from the data schema at run-time. It includes also the test server IDL for the description of the test component that resides on the CORBA APIs. This point is discussed in Section 4.3.

4.1 Data structure

As shown in Figure 3, the PDU IDL is generated from the test server IDL and message header IDL. The generator is called *PDU-Gen*. The test server IDL is designed by the test developer according to the test purposes, e.g. to verify the transport of different IDL typed values. The message header IDL consists of the modules GIOP and IOP defined in the CORBA specification.

The following is an excerpt that describes the structure of a GIOP 1.2 message header:

```
// IDL
module GIOP {
   struct Version {
      octet major;
      octet minor;
   };
   struct MessageHeader_1_2 {
      char          magic[4];
      Version       GIOP_Version;
      octet         flags;
      octet         message_type;
```

```
    unsigned long message_size;
  };
  struct RequestHeader_1_2 {
    // …
  };
```

```
    // …
};
```

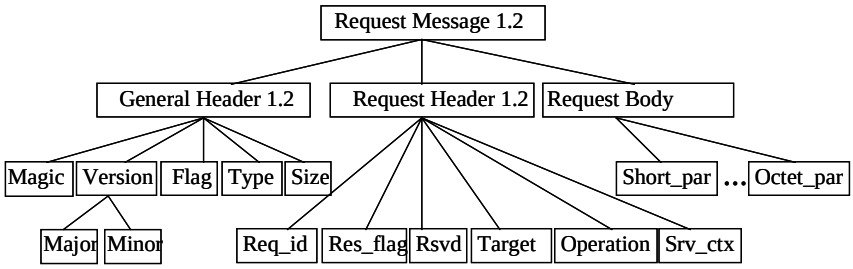

Figure 4 Request message structure

For the current GIOP version, eight messages (PDUs) are defined: Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError and Fragement. A GIOP PDU is composed of a general header, a specific header and a specific body. For example, a Request PDU consists of an instance of `MessageHeader_1_2`, followed by an instance of `RequestHeader_1_2`, followed by an instance of the Request body that consists of a sequence of the actual parameters. The one-the-wire representation is determined by applying the *Common Data Representation* (CDR) rules to the sequence of those IDL definitions. Figure 4 gives a graphical depiction of the structure of a Request PDU according to GIOP 1.2.

The various parts of the PDU are described through nested definitions of IDL-structs. Following the general GIOP header are the request header and body. The Request body contains the actual parameters of that particular operation. It consists of the CDR encodings of all the input and input/output parameters as they occur from left to right in the signature of that operation. The precise structure of the Request body depends on the signature of the test server interface and is only defined as a sequence of octets in the GIOP specification. The same holds true for the body of Reply-PDUs, which contains all the output parameters of an operation.

Since large portions of a GIOP PDU are already described in IDL, we use IDL as the language to describe the PDU structure for test cases.

Consider the following example:

```
interface TestServer {
  void foo (in short x, inout long y,
            out double z);
  // …
};
```

The operation `foo` has one input parameter of type `short`, one input/output parameter of type `long` and one output parameter of type `double`. The body of the Request-PDU will contain the `short` and `long` parameters and the body of the Reply PDU will contain the `long` and `double` parameters respectively (an input/output parameter is transferred in both directions between client and server). Using IDL, the structure of the bodies of the Request and Reply PDUs can be described as follows:

```
struct foo_Request_body {
  short x;
  long  y;
};
struct foo_Reply_body {
  long   y;
  double z;
};
```

The CDR is used again to marshal the actual parameters for the Request and Reply bodies. GIOP-Replies make a distinction between normal and abnormal execution that is indicated through an exception. This information can be coded in a similar way as shown above for normal execution.

For each of the operations of the test server interface, several structs are derived that each describe the content of a specific GIOP PDU. For example, the operation `foo` mentioned above yields the following structs:

```
struct foo_GIOP_Request {
  GIOP::MessageHeader_1_2 giop_header;
  GIOP::RequestHeader_1_2 request_header;
  foo_Request_body        request_body;
}
struct foo_GIOP_Reply_normal {
  // …
};
struct foo_GIOP_Reply_exception {
  // …
};
```

Repeating this scheme, all operations of the test server interface can be translated into a set of IDL specifications that describe the logical structure of all the GIOP PDUs that will be exchanged between the test client and the OUT.

4.2 Data schema

Since the generated PDU struture definitions, as described in the previous section, are used by test cases to get access to the database at run-time, a data schema that is in-line with the IDL definitions eases both the specification of constraints and the implementation of data access.

However, the PDU IDL is not directly used to derived the data schema. Because some information in the input IDLs which is not direct relevant for the test case specification, e.g. the attribute of an operation (normal or oneway), or the relations between message bodies of the same operation, is no more contained in the generated PDU IDL. Therefore, the generator for the data schema *Schema-Gen* takes the original IDLs as input.

Further, using an IDL-related language would be beneficial, because only minimal mappings of language constructs are required. We propose to use the Object-Definition Language (ODL) defined by the Object Database Management Group (ODMG). ODMG 3.0 [1] is the most recent specification of this consortium. It defines an object database framework, containing the data schema language ODL and the Object Query Language (OQL).

ODL provides *interface* and *class* types for objects. Interface types are used for generalization. Class types are used to directly instantiate objects. An object has an identifier and name, which can be used to refer to each other. ODL provides notations to specify relationships between objects. In particular, the referential integrity is guaranteed.

ODL is designed to be a super-set of IDL. Therefore it supports most of the basic types of IDL. In addition, collection objects are supported by the so-called type generators: `set`, `bag`, `list`, `array` and `dictionary`. Elements of collection objects are of the same type. Mostly used are `set` objects - collections of unordered, non-duplicated elements, and `list` objects - ordered collections of objects.

The data schema specification using ODL is guided by the following goals:

- Representing the complete information contained in the original test server IDL and the message header IDL.
- Taking care of aggregation relations and other dependence relations between data.
- Building data objects in a modular and hierarchical manner to allow flexible use.

As the GIOP interface provides the transparent communication between CORBA clients and servers, it is adequate to store values needed both by the test client and the test server in PDU structures.

The database does not store complete PDUs, but data objects in a hierarchical structure that allows flexible combination of PDU fields. The top-level structures are general message headers, specific message headers and specific message bodies, but not structures for complete PDUs such as `foo_GIOP_Request`. It reduces redundancy of storage space. Using constraints, PDUs can be constructed on-the-fly.

In IDL, a PDU part, e.g. `MessageHeader_1_2`, is defined by a `struct` type. Its fields are represented by `struct` members. There is an aggregation or composition relation between a `struct` type and its members. Each member may have different values. From testing perspective, these values may be valid (defined) or invalid (undefined or forbidden). In addition, there can be also relations between members of a structured type.

Therefore we propose the following IDL to ODL mappings (see also Figure 5):

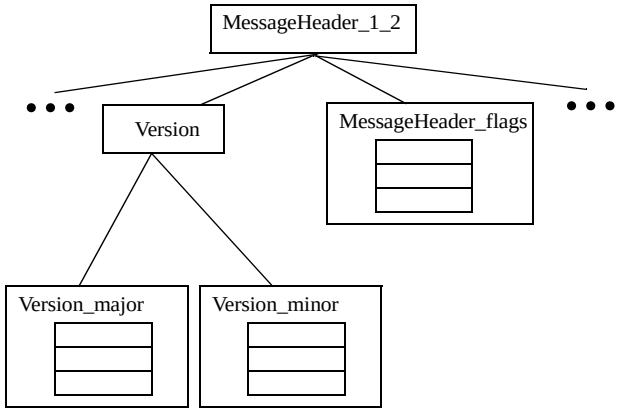

Figure 5 Aggregation relation

- Each value used in PDU parts is represented by a separate object instance. Data object inherits from a predefined interface named `DataObj`. The interface contains an attribute `flag` of type `octet`, to denote the usage of the value, e.g. valid or invalid for simple fields, normal or exceptional for message bodies.
- An IDL `struct` type is mapped to an ODL `class` type.
- Members of the struct type are mapped to attributes of the class type.
- For each member, a separate class type is defined.
- A basic type member is mapped to a class type that contains a `list` type attribute named `valuelist`. The type of the list is defined by a class that inherits from the interface `DataObj`, which contains a `value` attribute of the original type of the `struct` member.

- A member of the `struct` type that is itself a structured type, e.g. `struct` or `union`, is mapped to an attribute of the `class` type defined for the `struct` member, e.g. `Version`.

Using these rules, the IDL for message header from Section 4.1 can be translated into the following ODL:

```
module ODMG_GIOP {
   interface DataObj {
      octet flag;
   }
   class MessageHeader_1_2 : DataObj {
      MessageHeader_magic magic;
      Version GIOP_version;
      MessageHeader_flags flags;
      MessageHeader_message_type
         message_type;
      MessageHeader_message_size
         message_size;
   };
   class Version : DataObj {
      Version_major major;
      Version_minor minor;
   };
   class Version_major : DataObj {
      list<Version_major_value> valuelist;
   };
   class Version_major_value : DataObj{
      octet value;
   }
   class RequestHeader_1_2 : DataObj {
      //...
   };
   //...
};
```

That each value instance is accessible by referencing object instances is important for building dependence relations between data, as further discussed below.

Figure 6 illustrates a simplified view on the dependence relations (represented by arrows with dotted lines) in a Request message. The complete structure is depicted in Figure 4.

Within the Request header, the values of fields `Operation`, `Resp_flag` and `IOR` of Target (actually the type id of the operation interface contained in the IOR) are dependent from each other. The dependence is determined by the signature of the operation defined in the test server IDL. Also, the relation between `Operation` value and the corresponding Request body is defined by the operation signature, e.g. `op1` and `op1_Req_body`.

On the other hand, the relations between message bodies for a given operation, are determined by the semantics of the test server interfaces.

In ODL, the identified dependence relations are defined by `relationship` properties. For Figure 6, we may have the following ODL definitions, in which the class

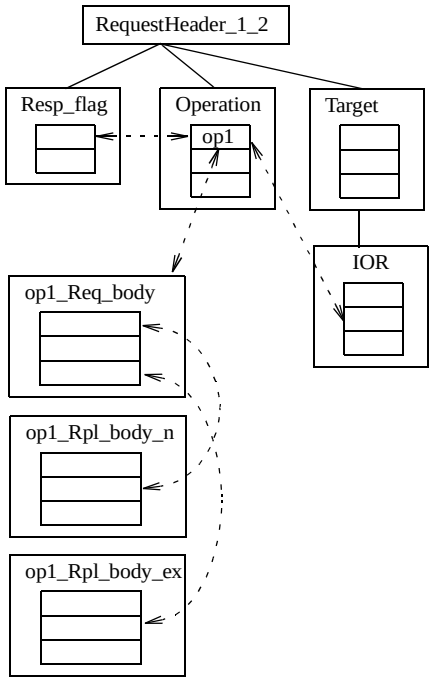

Figure 6 Dependence relations

`ReqHdr_operation_value` and the class `ReqHdr_resp_flag_value` reference to each other by bi-directional relationship properties. A pre-defined interface `RequestBody` is used by specific Request bodies to refer to corresponding operation names.

```
module ODMG_GIOP {
   interface RequestBody {
      relationship ReqHdr_operation_value op
         inverse ReqHdr_operation_value::req_bd;
      relationship ReplyBody_normal rpl
         inverse ReplyBody_normal req;
      relationship ReplyBody_exception ex
         inverse ReplyBody_exception req;
   };
   interface ReplyBody_normal { //...};
   interface ReplyBody_exception {//...};
   class RequestHeader_1_2 : DataObj {
      ReqHdr_request_id request_id;
      ReqHdr_resp_flags resp_flags;
      ReqHdr_reserved reserved;
      TargetAddress target;
      ReqHdr_operation operation;
      ODMG_IOP::ServiceContextList
         service_context;
   };
   class ReqHdr_resp_flag : DataObj {
      list<ReqHdr_resp_flag_value> valuelist;
   };
   class ReqHdr_resp_flag_value : DataObj {
      octet value;
      relationship ReqHdr_operation op
         inverse ReqHdr_operation::flg;
```

```
   };
   class ReqHdr_operation : DataObj {
      list<ReqHdr_operation_value> valuelist;
   };
   class ReqHdr_operation_value : DataObj {
      string value;
      relationship ReqHdr_resp_flag_value flg
         inverse ReqHdr_resp_flag_value op;
      relationship RequestBody req_bd
         inverse RequestBody::op;
   };
   class op1_Req_body : DataObj, RequestBody {
      //...
   };
   //...
};
```

4.3 Test cases

The test case specification is the second major part of a test specification. The test case behaviour describes the exchange of test events with the OUT. Since GIOP is a message-based protocol, messages are exchanged only as test events. In black-box testing, the basic principle of testing is that the test case sends a stimulus to the OUT, awaits in general a set of possible reactions from the OUT (including also unexpected ones), compares received reactions (including also the absence of a reaction) with the expected ones, and decides on the basis of this comparison how to proceed with the test cases or which verdict to assign (and to terminate the test case). The timing for the exchange of messages is controlled with timers: that a reaction from the OUT shall occur within a certain amount of time, that a reaction from the OUT shall be delayed by a certain amount of time, or that no reaction shall occur at all.

In the past, a methodology and framework for testing distributed systems, known as *conformance testing methodology and framework* (CTMF), have been developed and internationally standardized. CTMF covers all aspects of testing distributed systems such as test suite specification, test notation (TTCN - Tree and Tabular Combined Notation), test implementation and test execution. CTMF has been successfully applied to a number of systems which cover the range of e.g. e-mail systems and directory services to management and IN systems.

Currently, the third edition of TTCN (TTCN-3) [2] has been developed to address testing needs of modern telecom and datacom technologies and to widen the scope of applicability. TTCN-3 is a text-based language for the specification of tests for reactive systems in general.

TTCN-3 is on syntactical (and methodological) level very different to previous TTCN versions. However, the main concepts of TTCN have been retained and improved and new concepts have been included, so that TTCN-3 will be applicable for a broader class of systems. New concepts are, e.g. a test execution control program to describe relations between test cases such as sequences, repetitions and dependencies on test outcomes (see also Section), dynamic concurrent test configurations, and test behaviour in asynchronous and synchronous communication environments. Further improved concepts are, e.g. the integration of ASN.1, the module and grouping concepts to improve the test suite structure, and the test component concepts to describe concurrent test setups.

A test specification in TTCN-3 is included in a module, which declares all objects such as types, timer, test components, etc. used in testing and which in particular defines the test cases and their execution. The `GIOP-Tests` module described below imports data type definitions in IDL for test server and GIOP PDUs, as introduced in Section 4.1.

For the GIOP tests we use a test configuration with three test components (see also Figure 7): a test component for the vertical interface (TestClient), a test component for the horizontal interface of GIOP (TestServer), and a master test component (MTC) for the overall control of the test execution and coordination between the other two test components. The interface of the OUT is defined by a separate component type definition for OUT containing the vertical and horizontal GIOP interfaces.

```
// TTCN-3
module GIOP-Tests {
   import all from GIOP language IDL;
   import all from TestServer language IDL;
   type component MTC {
      port Coord CP;
   }
   type component TestClient {
      timer T_Response:= ...;
      timer T_NoResponse:= ...;
      timer T_Wait:= ...;
      port Coord CP;
      port VerGIOP VG;
   }
   type component TestServer {
      timer T_Response:= ...;
      timer T_NoResponse:= ...;
      timer T_Wait:= ...;
      port Coord CP;
      port HorGIOP HG;
   }
   type component OUT {
      port HorGIOP HG;
      port VerGIOP VG;
   }
   ...
}
```

The master test component executes the test cases. It uses the component type definition `MTC` (keyword `runs on`) and can be executed for `OUT`, which have interfaces according to the `OUT` component type definition (keyword

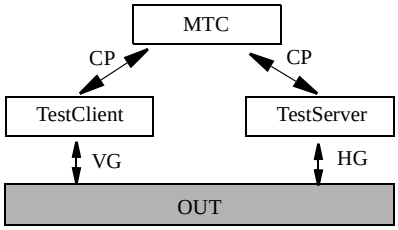

Figure 7 Test component configuration

system).

Initially, it creates the other two test components TC and TS, activates a default Default_1 to capture unexpected events during test execution, starts TC and TS with test behaviour, e.g. ReqMessage_T1_TC for TC (see also below), and awaits the termination of the created test components (all.done is blocking until all test components have terminated) in order to collect their individual test verdicts and to assign the overall test verdict. Test verdicts are pass (if the observed behaviour validates the test purpose of a test case), fail (if the observed behaviour disproves the test purpose of a test case), inconclusive (if the observed behaviour leads neither clearly to pass or fail), or error (for the exceptional cases of run-time errors in the test system).

Verdicts are collected by a TTCN-3 specific mechanism: each test component (i.e. all three in our case) have a local test verdict. This verdict is treated according to the "get-only-worse" rule: whenever during the execution of this test component an inconclusive or fail is assigned, the overall verdict of the test cases is at most inconclusive or respectively fail.

```
testcase ReqMessage_T1
(fRequest: foo_GIOP_Request,
 fRequestReply: foo_GIOP_Reply_normal, ...)
runs on MTC
system OUT
{
   var TestClient TC := TestClient.create;
   var TestServer TS := TestServer.s;
   connect(mtc:CP, TC:CP);
   connect(mtc:CP, TS:CP);
   map(TC:HG, system:HG);
   map(TS:VG, system:VG);
   activate(Default_1);
   TC.start(ReqMessage_T1_TC
   (fRequest, fRequestReply));
   TS.start(...);
   all.done;
   log ("successful termination);
   stop;
}
```

The behaviour of the test components TestClient and TestServer are defined in terms of functions, for instance, ReqMessage_T1_TC (see below). After activation of a default (in order to make the test behaviour robust for unexpected responses from the OUT), a request message fRequest is sent to the OUT via port VerGIOP. A timer is started in order to prevent from infinite waiting. In an alternative statement (keyword alt), the various responses are expected at port VerGIOP. Only in the case that the response is correct, a pass will be assigned. In the other cases (i.e. wrong response, e.g. different to the template fRequestReply, or no response, e.g. timeout), a fail will be assigned.

```
function ReqMessage_T1_TC
(fRequest: foo_GIOP_Request,
 fRequestReply: foo_GIOP_Reply_normal)
runs on TestClient
{
   activate (Default_2());
   VerGIOP.send(fRequest);
   T_Response.start;
   alt {
   [] VerGIOP.receive(fRequestReply)
      { set.verdict(pass); ... }
   [] VerGIOP.receive
      { set.verdict(fail); stop; }
   [] T_Response.timeout()
      { set.verdict(fail); stop; }
   }
}
```

The control part is used to denote the successive execution of test cases, to define the parameterization of test cases (using the data access described in the previous section) and to make the execution of test cases dependent on the outcome of previous test cases.

While message types (the structure for a test event exchanged between test system and OUT) are included from imported PDU IDL definitions, message templates (the concrete values or value constraints for a test event) are specified as the following examples foo_req_v and foo_rpl_v show.

A template definition alignes to the PDU IDL definition. It may use concrete values or symbolic values: any refers to all available values for the appropriate field; any is further constrainted by a flag which can be valid or invalid, normal or exceptional. The keyword dep is used to denote dependencies between templates.

The templates are used by external functions e.g. get_fooRequestMessage to get access to the database.

```
template foo_GIOP_Request foo_req_v
   := {
   giop_header any.valid,
   request_header any.valid,
   request_body.x any.normal
   }
template foo_GIOP_Reply_normal foo_rpl_v
   := dep foo_req_v {
```

```
   giop_header any.valid,
   reply_header any.valid,
   reply_body_normal any.normal
   }

external function get_fooRequestMessage
   (in template,
   out foo_GIOP_Request)
   return Boolean;

external function get_fooReplyNormalMessage
   (in template,
   in foo_GIOP_Request,
   out foo_GIOP_Reply_normal)
   return Boolean;

control {
   var foo_GIOP_Request curReq;
   var foo_GIOP_Reply_normal curRpl;
   while (get_fooRequestMessage
      (foo_req_v,curReq)) {
      get_fooReplyNormalMessage
      (foo_rpl_v,curReq, curRpl);
      execute(ReqMessage_T1
      (curReq, curRpl,...),20);
   }
}
```

The test case `ReqMessage_T1` is executed with different request and reply message as long as the data pool provides another request message for this test cases. The current data for a test case are retrieved via external functions `getRequestMessage` and `getRequestReplyNormalMessage`, respectively. `getRequestMessage` returns true if another message template is available. The execution of the testcase is defined in the `execute` statement. There, the concrete data is bound to the test case. The execution of the test case is limited to 20 seconds only. If the overall verdict cannot be assigned within that time, an error verdict will be assigned instead.

5. Concepts of test implementation

population of database
data access
generic client and server

6. Conclusions

completion and formalization of the IDL-ODL mappings.
example for use of database.
plan for realization.

7. References

[1] R.G.G. Cattell, et al.: The Object Database Standard: ODMG 3.0, Morgan Kaufmann Publ., Inc., 2000.

[2] ETSI: Methods for Testing and Specification (MTS): The Tree and Tabular Combined Notation version3 (TTCN-3), Core Language, Oct. 2000.

[3] Object Management Group: Common Object Request Broker Architecture (CORBA), ver. 2.4, Feb. 2001.

[4] The Open Group: CORBA Verification Suite, User's Guide, ver. 1.1.1, Sep. 1999.