

VSORB TEST SUITE RELEASE 1.0.0

TEST SUITE SPECIFICATION

ISSUE 1.0 - April 28, 1997

ISSUE HISTORY

ISSUE NUMBER	REASON FOR ISSUE
1.0	Initial Release

DISTRIBUTION
XoCorval

Produced for X/Open Company Limited by:

ApTest Ireland Limited
52 Lower Leeson Street
Dublin 2, Ireland
Phone: 353 1 662 9012
Fax: 353 1 662 7102
aptest@aptest.com

Comments on this document may be sent to xocorval@opengroup.org.

Motif, OSF/1 and UNIX are registered trademarks and X/Open, the “X Device” and The Open Group are trademarks of The Open Group.

Windows NT is a registered trademark of Microsoft Corporation.

CONTENTS

OBJECTIVE OF THE DOCUMENT	1
SCOPE OF THE DOCUMENT	1
ASSOCIATED REFERENCE DOCUMENTS	1
FUNCTIONALITY	2
Objective	2
Intended Users	2
Scope	3
Operational Interface	3
Host Environment	4
Network Environment	4
Utility to developers	4
COVERAGE	5
CORBA Reference Specification	5
CORBA Coverage	5
Interoperability	5
Level of testing	6
ARCHITECTURE	7
Components	7
Structure	7
Implementation Methods	10
Usage Models	12
Language Mapping Tests	13
IDL Compiler Tests	14
Stub Tests	15
CORBA API Tests	16
Interoperability Tests	27
PACKAGING	32
INSTALLATION AND CONFIGURATION	32
Installation	32
Configuration	32
REPORTING FACILITIES	33
DOCUMENTATION	34
GLOSSARY OF TERMS	34

1. OBJECTIVE OF THE DOCUMENT

This document provides a description of Release 1.0.0 of the VSORB Test Suite.

2. SCOPE OF THE DOCUMENT

This document gives an overview of VSORB in terms of:

- functionality,
- coverage,
- architecture,
- packaging,
- installation and configuration,
- reporting facilities, and
- documentation.

The target audience for this document is prospective licensees of the test suite seeking an introduction to its capabilities. Readers will benefit from a general familiarity with the governing CORBA specification, the TETware test harness, and the ADL test generation system.

3. ASSOCIATED REFERENCE DOCUMENTS

The CORBA specification defines the requirements tested by VSORB.

- 1) *The Common Object Request Broker: Architecture and Specification*. The Object Management Group.

Readers desiring detailed operational or implementation information regarding VSORB should refer to the documentation which accompanies the test suite.

- 2) *VSORB User's Guide*. X/Open Company Limited.
- 3) *VSORB Programmer's Guide*. X/Open Company Limited.

The TETware test harness provides the operating environment for VSORB.

- 4) *TETware User Guide*. X/Open Company Limited.

The Assertion Definition Language is used in developing the VSORB tests.

- 5) *ADL Translator User's Guide*. Sun Microsystems.

4. FUNCTIONALITY

4.1 Objective

In the CORBA model an application program operates in a distributed environment, composed of client programs and server objects cooperating through the medium of an Object Request Broker (ORB). The charter of VSORB is to test products that provide this ORB functionality for conformance with the CORBA specification.

VSORB verifies the functionality offered by an ORB to both clients and objects in order to facilitate three important areas of CORBA application portability:

- An ORB allows CORBA-conforming clients to be built and executed
- An ORB allows CORBA-conforming objects to be built and, when accessed by a client, executed
- An ORB will interoperate with other conforming ORBs, allowing clients to transparently access objects in a multi-vendor, multi-ORB environment

Prior to the advent of VSORB, most ORBs offered CORBA defined capabilities but with lots of variation in their interface and functionality. Hence clients and objects needed to be ported in order to use CORBA functionality on different ORBs. VSORB ensures consistent interfaces and functionality for CORBA-defined ORB features. Thus, applications can use these features without change across CORBA-conforming ORB implementations.

VSORB, like CORBA, focuses on interfaces and functionality rather than implementation. Thus ORB developers retain the flexibility to optimize and differentiate their ORBs for performance, operating environment, marketplace, etc., while providing a common operating interface to application clients and objects.

4.2 Intended Users

VSORB is designed for two primary uses:

- A branding program under the auspices of X/Open in which ORB implementations are tested for CORBA conformance and interoperability under formal processes and procedures.
- CORBA compliance testing by ORB implementors during product development and Quality Assurance.

4.3 Scope

The CORBA requirements tested by VSORB can be divided into five basic groups:

- C and C++ language mappings for CORBA APIs are correct
- IDL to C and C++ compilers accept IDL syntax and perform correct translations
- Requests can be made through translated code for C and C++
- CORBA APIs operate as specified from C and C++, including that requests can be made using the DII/DSI interfaces
- IIOP messages are accepted and generated with correct syntax and semantics

4.4 Operational Interface

VSORB is implemented under the TETware test harness, a version of the Test Environment Toolkit (TET), a widely used framework for implementing test suites. TETware provides a common environment for test users and developers and allows tests from different sources to be easily integrated together.

TETware consists of two major components:

- Libraries of functions for test implementation. These libraries are linked with VSORB's tests when they are built.
- The Test Case Controller (TCC) - an application the user runs to operate test suites which use the TETware libraries. The VSORB tests are built and executed using the TETware TCC.

The TCC provides simple commands for running VSORB and allows reports to be generated which summarize the results of testing. The operational capabilities the TCC offers include:

- build and run VSORB in a single automated step
- re-run only those tests which failed in a previous run
- run only an individual test
- optional progress reporting during test execution
- optional timeout on the duration of an individual test
- trace and debugging facilities

TETware is implemented on both UNIX and Windows NT systems, facilitating the use of VSORB in both these environments.

4.5 Host Environment

VSORB is designed to operate on systems compliant to the X/Open XPG3 and XPG4 specifications (e.g. most UNIX-based systems) and on Windows NT. An ISO C compilation system (header files, compiler and libraries of support functions) is required, and a C++ compilation must also be supplied if the C++ language binding is to be tested.

A Windows NT system must also provide the MKS Toolkit.

4.6 Network Environment

VSORB testing is generally performed using a single ORB and a single host system. Testing can also be performed using ORBs on multiple systems across a network. In fact this is transparent to VSORB, as it would be to any CORBA-compliant application.

4.7 Utility to developers

A number of VSORB features are of particular use in troubleshooting conformance problems during development and in regression testing:

- The ability to invoke tests individually
- Verbose journal file messages documenting test methods
- Detailed journal file messages documenting the expected and observed behavior for test failures
- Extensive configurability for implementation-specific behavior
- Resilience in the face of implementation problems
- Fully automated execution

The *VSORB User's Guide* contains information on troubleshooting common conformance issues and the *VSORB Programmer's Guide* contains information on the structure and implementation of the test suite to assist programmers needing to extend or enhance VSORB.

5. COVERAGE

5.1 CORBA Reference Specification

VSORB verifies the functionality of version 2.1 of the CORBA specification¹.

5.2 CORBA Coverage

VSORB covers the following CORBA requirements:

- IDL Syntax and semantics
- Dynamic Invocation Interface (DII) APIs
- Dynamic Skeleton Interface (DSI) APIs
- Interface Repository APIs
- ORB APIs
- Basic Object Adaptor (BOA) APIs (some interfaces only)
- Compiler testing for C/C++
- C Language Mapping (Version 1.1)
- C++ Language Mapping (Version 1.1)
- Interoperability (IIOP) (Version 1.1)

Other areas of CORBA are not covered in VSORB:

- Interoperability (DCE ESIOP)
- Interworking (CORBA/COM)
- Smalltalk, Ada, Java, and COBOL mappings
- Alternative Mappings For C++ Dialects
- BOA (some of the interfaces)

In some areas test coverage may be impacted by implementation-defined features.

In other areas test coverage is limited by testability limitations in the requirements defined by the specification. For example operations which are defined as freeing memory for a data object cannot be definitively tested since the memory allocation mechanism used is opaque.

5.3 Interoperability

VSORB can be used to test ORB interoperability in two ways:

- VSORB contains tests that explicitly verify an ORB's ability to

1. Actually the specification used is version 2.0 of CORBA, plus subsequent modifications to various parts of CORBA 2.0 by OMG's revision taskforces. These bits of specification are contributing elements of CORBA 2.1 so VSORB ends up covering the specification at the 2.1 level.

interoperate with other ORBs by exercising it can accept and generate IOP messages that pass requests between ORBs on behalf of a client. This entails use of VSORB components which simulate the role of a second ORB with which the ORB-under-test is communicating.

- VSORB's API and language mapping tests can also be used to drive interoperability testing involving multiple ORBs. This is accomplished by running VSORB with its objects installed on an ORB other than that with which its clients are communicating. This causes each VSORB test to invoke communication between the two ORBs to transfer information between the clients and objects. VSORB results should be identical in this configuration to those when it is used in testing with a single ORB.

Though this type of interoperability testing is not required for CORBA conformance testing, the thoroughness with which it exercises the CORBA interfaces makes VSORB very valuable in this role.

5.4 Level of testing

Three levels of testing coverage are applied in VSORB:

EXHAUSTIVE	Full testing of all possible aspects of a specification requirement. For example testing every possible value for an argument to an operation.
THOROUGH	Testing sufficient to validate that the implementation-under-test matches a specification requirement without exhaustive testing. For example, testing a subset of the possible values of an argument to an operation, exercising boundary and representative conditions.
IDENTIFICATION	Testing that the implementation-under-test provides the mechanism necessary to meet a specification requirement. For example, testing that an operation exists without validating its processing of arguments.

Exhaustive coverage is proved where practical, but in many areas exhaustive testing is not feasible due to the time and system resources required in executing such tests. Most VSORB testing is performed at the thorough level. This is also the case in most other industry test suites such as PCTS and VSX. While this represents some compromise of completeness in the testing performed, the compromise is fairly insignificant as implementations are rarely sensitive to the difference in coverage.

Identification testing is employed only where it is the only requirement provided by the governing specification, e.g. requiring that a request parameter be passed to an object but without specifying the contents of the parameter.

6. ARCHITECTURE

6.1 Components

VSORB consists of five basic elements:

- Interface definitions for test objects written in IDL
- Test clients written in C and C++
- Test object implementations written in C and C++
- Code which simulates a remote ORB for interoperability testing, written in C
- Report generators written in C

VSORB uses the first four of these elements in its tests. Three types of tests are performed:

- Tests in which the ORB's IDL compiler (for C or C++) is invoked to process the definition of the interface for one of the tests objects, in order to verify a capability of the IDL compiler
- Tests in which a test client causes the ORB-under-test to invoke one or more test objects, in order to verify the operation of a capability or interface of the ORB.
- Tests in which a simulated ORB communicates with the ORB-under-test using the IIOP protocol.

Report generators are used after VSORB tests have been run in order to produce reports based on the raw data emitted during test execution.

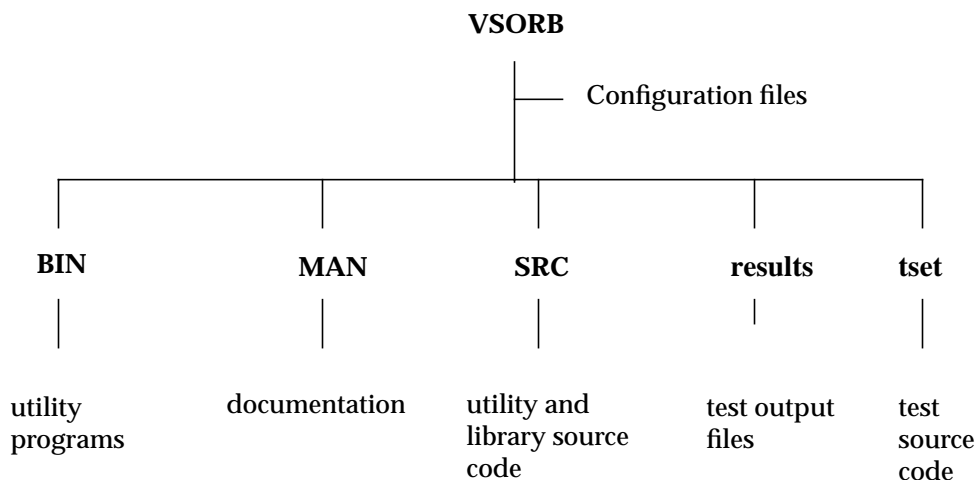
6.2 Structure

VSORB is a stand-alone test suite. Packaged and documented independently of other X/Open test suites, VSORB can be used in any appropriate host environment where TETware is installed.

TETware defines most of the structural aspects of VSORB, including its directory hierarchy, how tests are build and executed, how configuration information is maintained, and how test results are output and stored.

6.2.1 Directory structure

The high level directory structure of VSORB is shown in the following figure.



Test source code is grouped into sections under `tset` (i.e. separate directory trees for language mapping tests, IDL compiler tests, IIOP tests, etc.) and areas within each section (i.e. separate trees within the API test hierarchy for DII, ORB, BOA, etc.).

For each test area there are multiple independent tests. This decomposition of the test code helps in fault isolation. Each test is designed to analyze ORB behavior for a specific CORBA requirement and provide detailed failure information specific to that requirement if a conformance issue is encountered. As tests can be run individually, this allows each test failure to be separately diagnosed and resolved independently of running the entire test suite.

Code is shared across tests where practical with this code maintained in libraries in hierarchies under the `SRC` directory.

6.2.2 Building and Executing Tests

The three types of VSORB tests (for the IDL compiler, ORB APIs, and IIOP) are implemented in distinct fashions.

- Tests for the IDL compiler are shell scripts which employ standard shell utilities to perform testing and a set of shell APIs provided by TETware to report results.
- Tests for the ORB APIs are C and C++ functions which employ standard library functions to perform testing and a set of C and C++ APIs provided by TETware to report results.
- Tests for interoperability are also C and C++ functions which employ standard library functions to perform testing. However in addition to the TETware functionality used in the ORB API tests they also employ TETware's capabilities to operate distributed tests in which different test components are run on multiple host systems.

All three types of tests are executed under the control of the TETware TCC.

ORB and IIOP tests also include a Makefile which builds the test from source, linking the test with a TETware-supplied library into an executable program which is then run during testing. The Makefile also executes any commands necessary to establish preconditions for the test, such as running the IDL compiler to generate object stubs and skeletons if needed by the test. These build steps are also performed under the control of the TETware TCC, which performs the building of the VSORB tests in a single automated step, masking the details of the process from the user.

For example, to build and execute VSORB the user enters the following command

```
tcc -be VSORB all
```

6.2.3 Configuration information

Two types of TETware files control the operation of test suites under its control: *scenario files* and *configuration files*.

Scenario files generally do not need to be changed by the user. These files specify the test files to build and/or execute when the user invokes the TETware TCC. VSORB contains several scenarios, defining different sets of tests to be run depending on whether the C and or C++ language bindings are to be tested.

Configuration files contain information needed to adapt VSORB to a particular host environment and ORB. Configuration consist of the association of strings with arbitrary names, which generally start with the name of the test suite. For example:

```
VSORB_TEST_C_BINDING=Yes
```

is used to inform VSORB to test the C language binding.

Separate configuration files are provided for building the tests and for executing them.

- VSORB build configuration variables define the commands and options to be used in invoking the host C and C++ compilers to build the VSORB tests.
- VSORB execute configuration variables specify the behavior of the ORB in areas that are either outside the VSORB specification or which the specification allows to vary.

A third type of configuration mechanism in VSORB addresses the need to adapt VSORB to ORB- and system-specific behavior in areas likely to require executing some code, and which thus cannot be captured in a static configuration variable. This is handled by the definition of *user-defined routines* as part of the test suite. These routines are called when

needed by the test suite in order to perform an implementation-specific task, using code provided by the test suite user.

The calling interface and functional behavior required of these routines (which include shell scripts and C/C++ functions) is defined in the VSORB documentation along with instructions for the user on providing code which implements these requirements for a specific implementation. Sample routines are provided with VSORB for widely used implementations which can be used as is if appropriate, or as a model for developing a version suited to the ORB under test and the system where testing is being performed. Examples of user defined VSORB routines include:

- Implementing the system's security paradigm, e.g. granting a process the ability to perform a privileged operation
- Installing an object implementation
- Activating an object implementation
- Install an object in the Interface Repository

6.2.4 Test Reports

The output of executing VSORB is captured by TETware which combines the results and messages produced by all the tests run into a single *journal file*. This journal file can then be examined by the user or processed by report generators include in VSORB to produce various types of reports summarizing the results of testing.

6.3 Implementation Methods

VSORB tests are implemented in a combination of shell scripts and C and C++ source code which use TETware APIs for interfacing to the test journal and configuration files.

Tests are developed using standard modular programming methodology. As described above CORBA requirements are decomposed into many individual tests focused on specific conformance areas, sharing libraries of common code where possible. Similar methods are used in the object implementations used in the test suite. As well, objects and their implementations are used across multiple tests where this is practical.

C and C++ are used in the implementation of the VSORB tests in order to test the C and C++ language bindings defined by CORBA. To ensure each language binding is maximally exercised:

- Clients and objects written in C are used to test the CORBA APIs and the client stub mechanism under the C language binding.

- C++ versions of these clients and object used to test the C++ language binding.
- Where library code shared across tests performs non-ORB related functions (e.g. composing error messages) it is written in C.
- Any library code which is ORB-related is provided in both C and C++ versions.

Thus a C compiler is always required, regardless whether the C language binding is to be tested, but a C++ compiler is only required if the C++ binding tests will be run.

6.3.1 Use of System Functions

VSORB code is designed to be highly portable and operate without change across XPG and Windows NT systems. Most of the functions used by the C/C++ code are provided by VSORB and others are support library functions either defined in the ISO language specifications for C and C++ or known to be present on both XPG and Windows NT systems.

6.3.2 ADL

VSORB's tests for the CORBA APIs are developed using automatic programming tools rather than hand-coded as the other VSORB tests.

The Assertion Definition Language (ADL) used to develop the CORBA API tests is a system designed to produce tests from formal language definitions of programming interfaces. Interfaces to be tested are specified in ADL, a superset of IDL which captures the semantics as well as the syntax of an interface. An ADL translator parses these definitions and generates the code to test them.

Where tests need to perform interface-specific actions, to set up preconditions for calling the interface or for verifying its behavior, ADL generated tests call *auxiliary functions* which are written by the test suite developer. ADL automates writing much of the repetitive, and thus error prone code, that would also need to be hand generated otherwise. ADL also brings benefits to the maintenance of a test suite since the suite's source code can be modified at the ADL specification or auxiliary function level and the ADL translator run to regenerate new tests with generally less work and destabilization of the tests than would be possible otherwise.

In addition to generating tests for interfaces, ADL can also generate natural-language specifications for the interfaces by combining the ADL interface specification with a *Natural Language Dictionary* (NLD) provided by the test developer.

The tests generated by ADL are complete and ready to be run under TETware and the ADL translator does not need to be run in the process of building or executing VSORB.

VSORB also includes the ADL specification and NLD for the CORBA APIs to allow the user to examine and experiment with these materials and the ADL tools.

6.3.3 C++ Dialect

CORBA's language binding for C++ "assumes the target C++ environment supports all the features defined in the *Annotated C++ Reference Manual (ARM)* by Ellis and Stroustrup as adopted by the ANSI/ISO C++ standardization committee, including exception handling. In addition it assumes the C++ environment supports the namespace construct recently adopted into the language". However alternate mappings for other C++ dialects are also discussed in a CORBA Appendix, for environments lacking features such as exception handling, namespace, and RTTI.

VSORB's C++ testing does not verify these alternate dialects¹ - full ARM feature support as well as namespace is required for a C++ binding to pass the VSORB tests.

6.4 Usage Models

6.4.1 Troubleshooting Approach

The VSORB tests are implemented in a modular fashion designed to facilitate debugging by focusing tests on particular conformance issues. As well they are structured within VSORB's TETware scenario file so that tests for the simplest requirements are executed first, moving on to gradually more sophisticated requirements once more basic aspects of conformance have been established.

This simplifies the troubleshooting of subsequent tests which will most certainly fail if the features and interfaces tested earlier in the suite are not reasonably conforming.

6.4.2 Regression Testing

VSORB is primarily intended for conformance and development testing, however it is also useful in regression testing. In addition to providing a thorough measure of ORB behavior:

- Its tests provide repeatable methods with consistent results
- It includes a report generator to compare results of multiple runs

6.4.3 Stress Testing

The VSORB tests generate a substantial amount of activity for the ORB and are thus a useful stress tool in and of themselves. They can also be used in combination with other

1. One exception is the provision for 64-bit integers which is defined in the same CORBA Appendix as other dialect issues. VSORB allows for target C++ environments to support 64-bit integers and verifies they are supported as CORBA specifies.

stress-generating programs to ensure VSORB results do not change when additional activities are occurring on the ORB.

As well, the ADL-generated tests are driven by an ADL feature known as a *Test Data Dictionary* (TDD) which defines the values which will be used for parameters in testing the CORBA APIs. For conformance testing the TDD is set to test with representative and boundary values in order to achieve reasonable execution times in combination with thorough coverage. By extending the TDD to test over complete ranges of values for these parameters the user can then generate versions of these tests which perform extended, and thus more stressful, testing.

6.5 Language Mapping Tests

A basic element of a CORBA language mapping is a mechanism for programmers to access ORB functionality from their programming language. For C and C++ this includes expressing in a header file or files:

- Definitions of OMG IDL basic data types
- Definitions of OMG IDL constructed data types
- Constants defined in OMG IDL
- Signatures for the operations defined by the ORB, such as the dynamic invocation interface, the object adapter, and so forth

The role of this section of the VSORB tests is to verify the contents of these header files for the C and/or C++ mappings.

The tests consist of sets of C and C++ source files which include the implementation's header files for a CORBA language binding and reference the various elements which must be present therein.

The verification step for these tests is largely performed when the tests are built, as any missing elements or mismatches between the header files and the tests' expectations for the files' contents will result in compiler errors or warnings when the tests are built. These messages are captured in the journal file and result in a test failure. Some supplemental testing is performed at run time to check for requirements such as data type sizes.

These tests are executed first by the VSORB scenario file as ensuring correct header files is critical to building the test clients and objects used in later tests.

Testing is performed with many small test programs, each referencing a single header file element. In this way compiler messages specific to a particular requirement are emitted only for a specific tests focused on that requirement.

The CORBA requirements validated by these tests include:

- Names and sizes of basic and constructed data types
- Names of constants
- Names and signatures for CORBA API operations

6.6 IDL Compiler Tests

A second element of CORBA language mappings is the mapping of IDL constructs to a particular programming language.

The role of this section of the VSORB tests is to verify the IDL compilers for C and/or C++ implement CORBA's mappings for these languages.

The tests consist of sets of IDL source files which in sum cover all possible statement valid in OMG IDL, and a set of C and/or C++ source files which reference the data objects that an IDL compiler should produce from these source files.

The verification step for these tests consists of translating the VSORB IDL files with IDL compilers for C and/or C++ and then building the VSORB C and/or C++ source files, including in them the header files output by the IDL compilers.

Error and warning messages produced by an IDL compiler are indicative of a problem with the compiler. These messages are captured in the journal file and result in a test failure.

Similarly, error and warning messages from the C or C++ compiler while building the C or C++ test programs indicate that the mapping performed by the IDL compiler is not CORBA conforming. These messages are also captured in the journal file and result in a test failure.

These tests are executed second by the VSORB scenario file as correct IDL mapping is critical to building the test clients and objects used in later tests.

Testing is performed with many small tests programs, each addressing an element of IDL. In this way compiler messages specific to a particular requirement are emitted only for a specific tests focused on that requirement.

These tests cover the complete syntax of IDL as defined by the CORBA specification, including:

- Character set
- Identifier
- Literals
- Keywords

- Preprocessing
- Modules
- Types (basic, constructed, templates, etc.)
- Interfaces
- Inheritance
- Exceptions
- Operations
- Contexts
- Attributes

As well these tests apply this syntax in various permutations and combinations designed to test reasonable boundary conditions and combinations of language elements, for example nesting of types and name scopes.

6.7 Stub Tests

Where the Language Mapping and IDL Compiler tests described in the previous sections deal with the ability to build conforming clients and objects, these tests and those in the following section verify the ability to execute applications containing these clients and objects.

The Stub tests verify an application can make requests through the ORB using the interface descriptions an IDL compiler generates.

These tests consist of clients written in C and C++ which invoke the methods of various objects and verify parameters are passed between the client and object correctly.

The objects used are those whose IDL specifications were processed in the IDL Compiler tests as described above. Separate object implementations are provided in C and C++ and used for testing the respective language bindings.

Testing is performed using all valid parameter types, each as an in, out, and inout parameter.

Invalid requests are also made to invoke, where possible, system exceptions. As well the test objects raise user exceptions for some tests to ensure these are correctly communicated back to the client.

Each method used in these tests takes a special parameter. For each invocation the client places in this parameter a description of the type, value, and direction (in, out, in/out) of each other parameter along with directives to the object for what exception, if any, to raise and what values to return in any out or in/out parameters.

Upon receiving the request the object verifies the received parameters against the expected parameters. When returning the request the object returns a similar special

parameter which notes any mismatches between the actual and returned parameters. If no mismatch is found the object sets any out or in/out parameters to the values requested by the client, or if so requested raises an exception.

A problem reported by the object, a mismatch in returned parameters detected by the client, or a failure to communicate an exception raised by the object result in a test failure.

Both static and dynamic object invocation is tested in combination with the static requests made by the clients in these tests. The clients are run twice: communicating with test object implementations registered with the skeleton interface generated by the IDL compiler, and ones registered with a DIR interface to be invoked by the ORB through the DSI.

6.8 CORBA API Tests

These tests cover a variety of CORBA module operations¹ including the Dynamic Invocation Interface, Basic Object Adaptor Interface, ORB Interface, and Interface Repository Interface.

Tests in this section are generated with ADL as described above.

C and C++ clients are used for testing these operations in the respective language bindings.

In addition to those operations described in the following sections, which are tested for both the C and C++ bindings, the following operations are tested in just the C binding as they are not defined in the C++ binding.

Operations and requirements tested:

- *CORBA_exception_id*
 - return a pointer to a character string representing an exception
- *CORBA_exception_free*
 - free any storage associated with an exception (not definitively testable)
- *CORBA_exception_value*
 - return a pointer to a structure corresponding to an exception
- *CORBA_free*
 - free out parameter memory (not definitively testable)

1. Operation names used here are as per the C binding.

6.8.1 DII Operations

These tests verify that requests can be created and invoked dynamically.

These tests are similar to the stub tests described above. The same mechanism of passing a parameter describing the other arguments of a request is used and the tests are run against objects registered for both dynamic and static invocation.

As in the stub tests, combinations of parameter types and values are used.

The same object implementations are used in these tests as in the stub tests, however additional methods are also utilized to test the use of asynchronous and multiple request interfaces.

Associated operations for manipulating lists and context items are tested here as they are verified in part by including the data objects they manipulate in requests.

Operations and requirements tested:

- *CORBA_Context_create_child*
 - create a child context object
 - child is chained to parent
 - exceptions raised for error conditions
- *CORBA_Context_delete*
 - delete context
 - CTX_DELETE_DESCENDENTS flag semantics
 - exceptions raised for error conditions
- *CORBA_Context_delete_values*
 - delete context object properties
 - wild card character semantics
 - exceptions raised for error conditions
- *CORBA_Context_get_values*
 - retrieve context object properties
 - CTX_RESTRICT_SCOPE flag semantics
 - exceptions raised for error conditions
- *CORBA_Context_set_one_value*
 - sets a context object property
 - exceptions raised for error conditions
- *CORBA_Context_set_values*
 - set context object properties
 - exceptions raised for error conditions

- *CORBA_NVList_add_item*
 - add item to a list
 - IN_COPY_VALUE flag semantics
 - DEPENDENT_LIST flag semantics
 - exceptions raised for error conditions
- *CORBA_NVList_free*
 - free a list (not definitively testable)
 - exceptions raised for error conditions
- *CORBA_VList_free_memory*
 - free out_arg memory (not definitively testable)
 - exceptions raised for error conditions
- *CORBA_NVList_get_count*
 - return the number of items in a list
- *CORBA_Request_add_arg*
 - add arguments to a request
 - IN_COPY_VALUE flag semantics
 - exceptions raised for error conditions
- *CORBA_Request_delete*
 - delete a request
 - exceptions raised for error conditions
- *CORBA_Request_get_response*
 - determine if a request is complete
 - RESP_NO_WAIT flag semantics
 - exceptions raised for error conditions
- *CORBA_Request_get_next_response*
 - return the next request that completes
 - RESP_NO_WAIT flag semantics
 - exceptions raised for error conditions
- *CORBA_Request_invoke*
 - initiate a request synchronously
 - exceptions raised for error conditions
- *CORBA_Request_send*
 - asynchronously initiate a request
 - INV_NO_RESPONSE flag semantics
 - exceptions raised for error conditions

- *CORBA_Request_send_multiple_requests*
 - asynchronously initiate multiple requests
 - INV_NO_RESPONSE flag semantics
 - INV_TERM_ON_ERR flag semantics
 - exceptions raised for error conditions

6.8.2 DSI Operations

The DSI API specifies the format of the up-call made to an object that is invoked dynamically. This is tested in the stub and DII tests which validate requests are correctly passed to an object when it is invoked in this fashion. There are not separate tests in VSORB for this area of CORBA.

6.8.3 ORB and Object Operations

Tested by invoking each operation and verifying the results.

Operations and requirements tested:

- *CORBA_Object__duplicate*
 - duplicate is indistinguishable from the original
- *CORBA_Object__create_request*
 - resulting request invokes specified operation
 - specified arguments are passed
 - context is passed
 - OUT_LIST_MEMORY flag semantics (not definitively testable)
 - exceptions raised for error conditions
- *CORBA_Object__get_interface*
 - returns object from the Interface Repository
 - exceptions raised for error conditions
- *CORBA_Object__hash*
 - different results indicate references are not identical (not definitively testable)
 - result does not change over lifetime of a reference
 - maximum parameter specifies upper bound
- *CORBA_Object__is_a*
 - returns correct value for all types
 - returns correct value for ancestors

- *CORBA_Object__is_equivalent*
 - results for identical and non-identical references (not definitively testable)
- *CORBA_Object__is_nil*
 - indicate if the reference denotes no object
- *CORBA_Object__non_existent*
 - returns correct value for objects which do and don't exist (not definitively testable)
- *CORBA_Object__release*
 - storage for object reference is reclaimed (not directly testable)
 - other object references not affected
- *CORBA_ORB_list_initial_services*
 - returns ObjectIds appropriate to the implementation
- *CORBA_ORB_get_default_context*
 - returns default process context object
- *CORBA_ORB_object_to_string*
 - is inverse of *string_to_object*
- *CORBA_ORB_create_alias_tc*
 - TypeCode for an alias is created
- *CORBA_ORB_create_array_tc*
 - TypeCode for an array is created
- *CORBA_ORB_create_enum_tc*
 - TypeCode for an enumeration is created
- *CORBA_ORB_create_exception_tc*
 - TypeCode for an exception is created
- *CORBA_ORB_create_interface_tc*
 - TypeCode for an interface is created
- *CORBA_ORB_create_list*
 - list of specified size is created
 - list is cleared
 - items may be added to the list
 - items may be added by indexing into the list structure
 - exceptions raised for error conditions

- *CORBA_ORB_create_recursive_sequence_tc*
 - TypeCode for a recursive sequence is created
- *CORBA_ORB_create_sequence_tc*
 - TypeCode for a sequence is created
- *CORBA_ORB_create_string_tc*
 - TypeCode for a string is created
- *CORBA_ORB_create_struct_tc*
 - TypeCode for a structure is created
- *CORBA_ORB_create_union_tc*
 - TypeCode for a union is created
- *CORBA_ORB_create_operation_list*
 - list is created
 - list is initialized with argument descriptions for specified operation
 - arguments are in same order defined for the operation
 - exceptions raised for error conditions
- *CORBA_ORB__resolve_initial_references*
 - returns list of ObjectIds appropriate to the implementation
 - exceptions raised for error conditions
- *CORBA_ORB_string_to_object*
 - is inverse of *object_to_string*
 - generates a valid object reference

The operations not tested are: *CORBA_Object__get_implementation*.

6.8.4 BOA Operations

Tested by having object implementations invoke these operations and verifying the results.

Operations and requirements tested:

- *CORBA_BOA_create*
 - new object reference is created
 - exceptions raised for error conditions
- *CORBA_BOA_dispose*
 - object reference is invalidated

- *CORBA_BOA_get_id*
 - id value is returned
- *CORBA_BOA_set_exception*
 - various types of exceptions can be raised

The operations not tested are: *CORBA_BOA_change_implementation*, *CORBA_BOA_deactivate_impl*, *CORBA_BOA_get_principal*, *CORBA_BOA_impl_is_ready*, and *CORBA_BOA_obj_is_ready*.

6.8.5 Interface Repository Operations

Read operations are tested by retrieving interface definitions previously added to the Repository and verifying they are correct. Definitions are added through a user-defined routine as the write operations may not be meaningfully provided by all implementations. Write operations are tested by adding elements to a repository and retrieving and verifying them.

Operations and requirements tested are as follows. Read and write attributes of the relevant objects are tested in the process of verifying these operations:

CORBA_AliasDef

- Inherited CORBA_TypeDef operations

CORBA_ArrayDef

- Inherited CORBA_IDLType operations

CORBA_AttributeDef

- Inherited CORBA_Contained operations

CORBA_ConstantDef

- Inherited CORBA_Contained operations

CORBA_Contained

- Inherited CORBA_IRObjct operations

- *CORBA_Contained_describe*
 - returns a description of the interface
- *CORBA_Contained_move* (allowed to return NoPermission in all cases)
 - moves an object from one Container to another
 - exceptions raised for error conditions

CORBA_Container

- Inherited CORBA_IObject operations
- *CORBA_Container_contents*
 - return contained or inherited objects
 - return constants, typedefs and exceptions in an InterfaceDef and defined and inherited attributes and operations
 - limit_type selects interface type
 - limit_type = dk_all selects all
 - exclude_inherited controls if inherited objects are returned
- *CORBA_Container_create_alias* (allowed to return NoPermission)
- *CORBA_Container_create_constant* (allowed to return NoPermission)
- *CORBA_Container_create_enum* (allowed to return NoPermission)
- *CORBA_Container_create_interface* (allowed to return NoPermission)
- *CORBA_Container_create_module* (allowed to return NoPermission)
- *CORBA_Container_create_struct* (allowed to return NoPermission)
- *CORBA_Container_create_union* (allowed to return NoPermission)
 - creates the specified element as a contained object
 - exceptions raised for error conditions
 - if multiple versions are not supported, exception if the name already exists
- *CORBA_Container_describe_contents* (the *describe* operation on various objects is also exercised in the process of verifying this operation)
 - return descriptions of contained or inherited objects
 - limit_type selects interface type
 - limit_type = dk_all selects all
 - exclude_inherited controls if inherited objects are returned
 - max_returned_objects limits number of objects returned
 - max_returned_objects != -1 means no limit on number of objects returned
 - exceptions raised for error conditions
- *CORBA_Container_lookup*
 - locates definitions

- *CORBA_Container_lookup_name*
 - locates an object by name
 - levels_to_search = -1 searches current and contained objects
 - levels_to_search = 1 searches current object only
 - limit_type selects interface type
 - limit_type = dk_all selects all
 - exclude_inherited controls if inherited objects are returned

CORBA_EnumDef

- Inherited CORBA_TypeDef operations

CORBA_ExceptionDef

- Inherited CORBA_Contained operations

CORBA_IDLType

- Inherited CORBA_IObject operations

CORBA_InterfaceDef

- Inherited CORBA_Contained operations
- Inherited CORBA_Container operations
- Inherited CORBA_IDLType operations
- *CORBA_InterfaceDef_create_attribute* (allowed to return NoPermission)
- *CORBA_InterfaceDef_create_operation* (allowed to return NoPermission)
 - creates the specified element
 - exceptions raised for error conditions
- *CORBA_InterfaceDef_describe_interface*
 - returns a description of an interface and its attributes
- *CORBA_InterfaceDef_is_a*
 - true if interface inherits directly or indirectly from specified interface, false otherwise

CORBA_IObject

- *CORBA_IObject_destroy* (allowed to return NoPermission)
 - causes object to cease to exist

CORBA_ModuleDef

- Inherited CORBA_Contained operations
- Inherited CORBA_Container operations

CORBA_OperationDef

- Inherited CORBA_Contained operations

CORBA_PrimitiveDef

- Inherited CORBA_IDLType operations

CORBA_Repository

- Inherited CORBA_Container operations
- *CORBA_Repository_create_array* (allowed to return NoPermission)
- *CORBA_Repository_create_sequence* (allowed to return NoPermission)
- *CORBA_Repository_create_string* (allowed to return NoPermission)
 - creates the specified element
- *CORBA_Repository_getprimitive*
 - returns a reference to a PrimitiveDef for the specified kind
- *CORBA_Repository_lookup_id*
 - locates definitions in Repository
- *CORBA_Repository_lookup_name*
 - locates an object by name
 - levels_to_search = -1 searches current and contained objects
 - levels_to_search = 1 searches current object only
 - limit_type selects interface type
 - limit_type = dk_all selects all
 - exclude_inherited controls if inherited objects are returned

CORBA_SequenceDef

- Inherited CORBA_IDLType operations

CORBA_StringDef

- Inherited CORBA_IDLType operations

CORBA_StructDef

- Inherited CORBA_TypeDef operations

CORBA_TypeDef

- Inherited CORBA_Contained operations
- Inherited CORBA_IDLType operations

CORBA_TypeCode

- *CORBA_TypeCode_equal*
 - indicates if type codes are interchangeable
- *CORBA_TypeCode_kind*
 - indicates what operations can be invoked on a TypeCode
- *CORBA_TypeCode_Id*
 - returns the RepositoryId identifying the type
 - exceptions raised for error conditions
- *CORBA_TypeCode_name*
 - returns the name identifying the type
 - exceptions raised for error conditions
- *CORBA_TypeCode_member_count*
 - returns the number of members for the type
 - exceptions raised for error conditions
- *CORBA_TypeCode_member_name*
 - returns the name of the specified member
 - exceptions raised for error conditions

- *CORBA_TypeCode_member_type*
 - returns the type of the specified member
 - exceptions raised for error conditions
- *CORBA_TypeCode_member_label*
 - returns the label of the specified union member
 - exceptions raised for error conditions
- *CORBA_TypeCode_discriminator_type*
 - returns the label of the type of all non-default member labels
 - exceptions raised for error conditions
- *CORBA_TypeCode_default_index*
 - returns the index of the default member
 - returns -1 if no default member
 - exceptions raised for error conditions
- *CORBA_TypeCode_length*
 - returns the bound for strings and sequences
 - returns the number of elements for an array
 - exceptions raised for error conditions
- *CORBA_TypeCode_content_type*
 - returns the element type for sequences and arrays
 - returns the original type for an alias
 - exceptions raised for error conditions

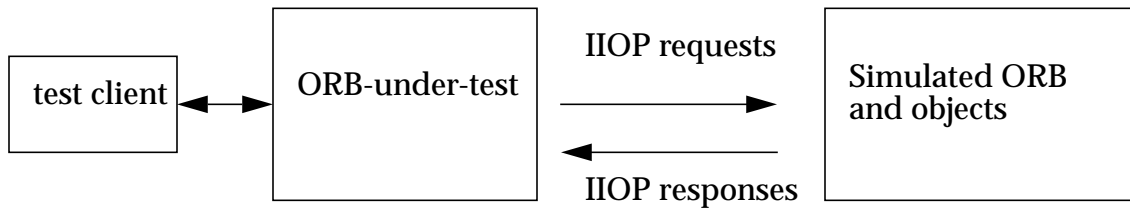
CORBA_UnionDef

- Inherited CORBA_TypeDef operations

6.9 Interoperability Tests

These tests verify that clients of the ORB-under-test can access objects of other ORBs, and objects of the ORB-under-test can be called by clients of other ORBs.

For verifying the ORB-under-test can access objects on other ORBs, VSORB employs code which simulates a second ORB and establishes a configuration in which simulated objects are located on this simulated ORB. During testing the simulated ORB appears to the ORB-under-test as a remote ORB with which it needs to communicate to access remote objects on behalf of its local clients. Test clients invoke the ORB-under-test to access the simulated objects and thus stimulate communication with the simulated ORB. The simulated ORB validates the messages it receives and issues appropriate replies. The test clients verify the responses received from the ORB-under-test as a result.

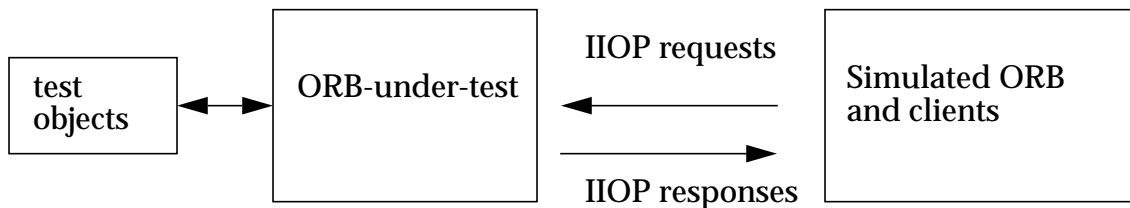


Testing issuing requests to a remote ORB

For verifying objects on the ORB-under-test can be accessed from other ORBs, VSORB employs code which simulates a second ORB and establishes a configuration in which test objects are located on the ORB-under test. During testing the simulated ORB appears to the ORB-under-test as a remote ORB communicating with it to access its objects on behalf of remote clients. The simulated ORB sends requests to the ORB-under-test. The test objects validate the calls they receive from the ORB-under-test as a result, and the simulated ORB validates the reply messages the ORB-under test sends it.

These tests are run twice, to ensure complete support for Interoperability throughout the ORB implementation:

- testing is performed using stubs for the simulated objects generated by the ORB-under-test's IDL compiler
- testing is performed using DII calls. Both asynchronous and synchronous invocations are tested



Testing issuing requests from a remote ORB

These tests employ the distributed features of TETware which allow pieces of a test to be executed on different host systems, with running the test elements and communication and synchronization between them managed by TETware.

TETware allows its logical local and remote system to be the same physical system. Thus the VSORB interoperability tests can be run using a single system on which both the ORB-under-test and the simulated ORB are installed, or in a network configuration in which the ORB-under-test and the simulated ORB are located on different systems.

A major focus of these tests is on verifying the syntax and semantics of the actual protocol messages involved in inter-ORB communication. The simulated ORBs construct and analyze IIOP messages byte-by-byte to ensure each aspect of the CORBA interoperability protocols are supported. This provides a more comprehensive test of an ORB's support of these protocols than can be provided by simply connecting two complete ORB implementations.

The coverage of these tests can be grouped as follows:

- CDR (Common Data Representation) encoding/decoding and encapsulations are correct
- IIOP messages are correctly accepted and generated
- GIOP message transfer assumptions are met
- IIOP 1.0 and (optionally) 1.1 protocol versions are supported

6.9.1 CDR formatting

These tests verify that all IDL data types are encoded and decoded correctly by the ORB-under-test. The tests cover encoding and decoding of primitive data types, alignment, byte ordering, IDL constructed types, pseudo object types, and object references.

CDR support is verified in four ways:

- Request messages are sent containing in parameters which must be decoded and passed to objects
- Request messages sent containing out parameters which must be decoded and then encoded in the corresponding reply messages
- Requests are invoked containing in parameters which must be encoded and passed to the simulated ORB
- Requests are invoked containing out parameters which must be encoded and then decoded from the corresponding reply messages sent by the simulated ORB

All the primitive data types and constructed types are tested:

- char
- octet
- short
- unsigned short
- long
- unsigned long

- float
- double
- boolean
- struct
- union
- array
- sequence
- string
- enum

The objects used in these tests include operations with combinations of those types as parameters.

Alignment is checked by combining different types of parameters.

The following pseudo object types are tested:

- TypeCode
- Any
- Exception
- Context

Covered typecodes are:

- tk_null
- tk_void
- tk_short
- tk_long
- tk_ushort
- tk_ulong
- tk_float
- tk_double
- tk_boolean
- tk_char
- tk_octet
- tk_any
- tk_TypeCode
- tk_objref
- tk_struct
- tk_union
- tk_enum
- tk_string
- tk_sequence
- tk_array
- tk_alias
- tk_except
- recursive TypeCode

IORs are tested in both initiating and receiving requests and through using an object as an operation parameter.

Principal pseudo objects not addressed in these tests. Principal values have no standard format or interpretation, beyond serving to identify callers. The CORBA specification does not define any inter-ORB security mechanisms, or prescribe any usage of Principal values.

6.9.2 IIOP Messages

Each of the seven IIOP message types are tested:

- Request
- Reply
- CancelRequest
- LocateRequest
- LocateReply
- CloseConnection
- MessageError

Request and Reply messages are tested in both directions. By making requests of objects on the ORB-under-test, its handling of incoming Request messages and its outgoing Reply messages are tested. By making requests of objects implemented on the simulated ORB, outgoing Requests from the ORB-under-test and its handling of incoming Reply messages are tested. Both Requests with responses expected and those without are tested.

There are four possible reply statuses to these messages:

- NO_EXCEPTION
- USER_EXCEPTION
- SYSTEM_EXCEPTION
- LOCATION_FORWARD

The first three types are tested in both directions. LOCATION_FORWARD is tested only as a Reply from the simulated ORB.

LocateRequest messages are tested one way only, since there is no way to control when and if an ORB will emit these message. The simulated ORB sends a LocateRequest message to the ORB-under-test, expecting a LocateReply from it.

CloseConnection is only sent from server ORBs to client ORBs. It is sent to the ORB-under-test by the simulated ORB.

CancelRequest is defined as advisory and thus is not definitively testable. The simulated ORB sends this request to the ORB-under-test to verify it will not reject it, but expects no particular behavior as a result.

MessageError is tested by sending various ill-formed messages from the simulated ORB to the ORB-under-test, and expecting this message in reply.

GIOP also supports passing a service specific context. This is not covered in VSORB as service-specific functionality is outside the scope of testing for CORBA.

6.9.3 Transport assumptions

A compliant ORB must support the multiplexing of requests over a single connection. Multiple independent requests for different objects or a single object may be sent over this connection. VSORB verifies conformance to these requirements by using a connection in both these fashions when making requests to objects of the ORB-under-test.

6.9.4 IIOP version support

The CORBA specification requires that IIOP 1.1 implementations also support the IIOP 1.0 protocol. VSORB provides tests for both levels of protocol and verifies the appropriate version(s) are supported by the ORB-under-test.

Testing specific to GIOP 1.1 includes:

- a revised message header format
- message fragmentation

7. PACKAGING

VSORB is packaged as a compressed tar file for distribution via the Internet.

All releases to include source to the VSORB tests as well as the ADL files used in generating VSORB tests.

8. INSTALLATION AND CONFIGURATION

8.1 Installation

Installing VSORB requires use of the `tar` command to install the test suite file.

VSORB contains several libraries and utilities that need to be built before tests can be built or executed. A Makefile is provided that is executed by the user after the test suite is “untar-ed” to perform this installation process.

8.2 Configuration

VSORB allows the user to specify a variety of information about the ORB being tested

and the system on which testing is to be performed. This extensive configurability allows VSORB to be used on a variety of systems and with a variety of ORB implementations.

Configuration information is read from TETware configuration files. Values for the variables defined in these files are set by the user prior to executing the test suite. Configuration file entries cover items such as:

- Whether an XPG or Window NT system is being used for testing
- The commands and options to be used when invoking the system compilers
- The commands and options to be used when invoking the ORB's IDL compilers
- Whether the C and/or C++ bindings are to be tested
- Whether the ORB supports allowable options in the CORBA specification
- How the ORB implements variations between implementations allowed by the CORBA specification, such as implementation defined limits and parameters

9. REPORTING FACILITIES

A run of VSORB produces a TETware journal file documenting:

- The tests run
- The test methods used
- The result of each test
- Details of the expected and observed behavior for all test failures

As this raw data file is quite large, a report generator is provided which extracts more user-friendly reports from these journals.

A report contains summary information on the "test run" including:

- The version of the test suite and specification involved
- Start and completion time
- System and OS name tested where testing was performed
- The number of tests executed, by section and in total
- The total number of each possible test result reported, by section and in total

A report can also provide additional detail for different types of test results, for example showing the output of only those test which failed during the run.

Two additional report generators are provided:

- A utility which generates a formal conformance report for use in X/Open branding.

- A utility which compares the results from multiple journal files. This utility is especially useful in regression testing to compare the VSORB results of different releases of a product.

10. DOCUMENTATION

VSORB includes a *User's Guide* providing information on operating the suite. The content of this guide includes:

- Installation
- Configuration
- Execution
- Report generation
- Debugging and trouble shooting

VSORB includes a *Programmer's Guide* providing information on maintaining and evolving the suite. The content of this guide includes:

- Source structure
- Theory of operation
- Implementation notes
- Procedures for adding additional tests

Each VSORB release also contains Release Notes documenting:

- Changes since the last release
- Known problems and work arounds

11. GLOSSARY OF TERMS

ADL	The Assertion Definition Language, an automated test generation system used in developing some of the VSORB tests.
Configuration File	A TETware file which contains the names of VSORB configuration variables and values assigned to them by the user for a specific ORB implementation and test system environment.
CORBA	OMG's <i>Common Object Request Broker: Architecture and Specification</i> .
VSORB	The VSORB Test Suite covered by this specification.
IDL	OMG's Interface Definition Language defined in the CORBA

specification.

ORB	An Object Request Broker as defined in the CORBA specification.
Journal File	A TETware journal is the file into which test results and tracking data are deposited by the TETware TCC.
NLD	The Natural Language Dictionary - a feature of ADL in which the test developer defines natural language equivalents to ADL statements describing a function's semantics to allow the ADL translator to produce a natural language specification for the function.
Scenario File	A TETware test scenario is a sequence of one or more invocable components associated with a single user-exposed name. There may be multiple test scenarios per scenario file. Test scenarios are used by the TCC to translate user requests into actions.
TDD	The Test Data Dictionary - a feature of ADL in which the test developer defines the values to be used for the parameters to a function when testing it.
TETware	A version of the Test Environment Toolkit, a user interface and programming environment for test suites, developed by X/Open.
User Defined Routine	A piece of user supplied code called by VSORB when it needs to perform an ORB- or system-specific operation.
XoCorval	An electronic mail group populated by the organizations involved in the development of VSORB.