Object-Oriented Perspective on Software System Testing in a Distributed Environment

A flexible object-oriented test system was developed to deal with the testing challenges imposed by software systems that run in distributed client/server environments.

by Mark C. Campbell, David K. Hinds, Ana V. Kapetanakis, David S. Levin, Stephen J. McFarland, David J. Miller, and J. Scott Southworth

In recent years software technology has evolved from single-machine applications to multimachine applications (the realm of the client and server). Also, object-oriented programming techniques have been gaining ground on procedural programming languages and practices. Recently, test engineers have focused on techniques for testing objects. However, the design and implementation of the test tools and code have remained largely procedural in nature.

This paper will describe the object testing framework, which is a software testing framework designed to meet the testing needs of new software technologies and take advantage of object-oriented techniques to maximize flexibility and tool reuse.

System Software Testing

The levels of software testing include unit, integration, and system testing. Unit testing involves testing individual system modules by themselves, integration testing involves testing the individual modules working together, and system testing involves testing the whole product in its actual or simulated operating environment. This paper focuses on software system testing.

A software system test is intended to determine whether the software product is ready to ship by observing how the product performs over time while attempting to simulate its real use. System testing is composed of functional, performance, and stress tests. It also covers operational, installation, and usability aspects of the product and may include destructive and concurrence testing. The product may support many different hardware and software configurations which all require testing. All of these aspects are combined to assess the product's overall reliability. Software system testing is usually done when all of the individual software product components are completed and assembled into the final product.

In the past, system testing environments centered around testing procedural, nondistributed software. These environments, which were also procedural and nondistributed, were usually developed by the test writer on an ad hoc basis along with the test code for the product. Recently, software system testing has benefited from the use of highly automated test harnesses and environments that simplify test execution, results gathering, and report generation (see Fig. 1). Unfortunately, the test harnesses created in these environments were not easily reusable, and when the next project reached the test planning stage, the test harness had to be reworked.

The advent of standardized test environments such as TET (Test Environment Toolkit)* helped to reduce this costly retooling by providing a standard API (application program interface) and tool base that test developers can adopt and use to write standardized tests. However, the difficulty is to provide a standard test harness that is complete but flexible enough to keep pace with changing software technology and remain viable for the long term.

During the development and testing of the initial release of HP ORB Plus, which is an object request broker based on the Object Management Group's CORBA specification (see page 76), we realized that distributed object technology posed testing problems that were not adequately solved by any of the test harnesses currently available. We needed a flexible test environment that could handle heterogeneous

* The Test Environment Toolkit (TET) specification began in September 1989 as a joint proposal by the Open Software Foundation, UNIX® International, and X/Open®.

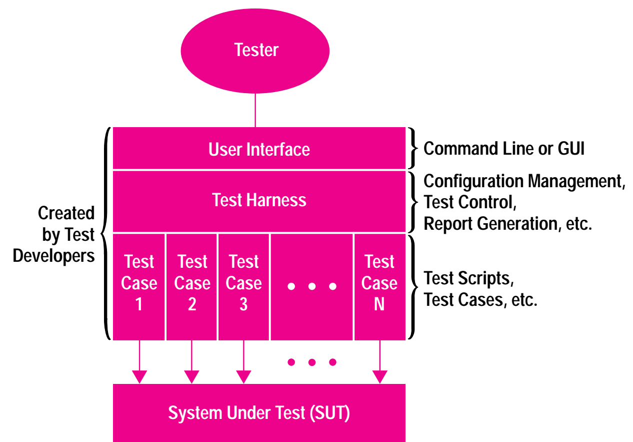

Fig. 1. A typical automated test environment.

distributed systems communicating over multiple transports using multithreaded clients and servers. However, we were not willing to lose the investment we made in the test code and tools developed for our earlier products.

Instead of abandoning the old test environment and replacing it with an entirely new system, we decided to use the object-oriented principles of encapsulation and polymorphism to evolve our current environment base to meet our needs without throwing out the old code. The ability to change or replace functional blocks of a system without affecting the entire environment is one of the main benefits of object-oriented design (see "Object-Oriented Programming" on page 79). Object-oriented principles allowed us to reuse existing tools.

Distributed System Testing

In a distributed object system, service providers are distributed across a network of systems and are called servers. Clients, who issue requests for those services, are also distributed across a network. A single program can be both a client (issues requests) and a server (services requests). Clients can issue requests to local and remote servers. During a distributed object system test, clients are responsible for reporting any failures or status resulting from the requests they make.

The first task performed during the system testing of a distributed object software product is test setup. Clients and servers must be deployed across the network to targeted systems. Consideration must also be given to the fact that servers may have multiple clients sending messages to them, and the distribution of clients and servers may change during a system test so that various hardware and software configurations can be tested.

The Object Management Group's Distributed Object Model

The Object Management Group (OMG) creates standards for the interoperability and portability of distributed object-oriented applications. The OMG only produces specifications, not software. Each participating vendor provides an implementation to meet the specification. The Common Object Request Broker Architecture (CORBA) specification defines a flexible framework upon which distributed object-oriented applications can be built. This architecture represents the Object Request Broker (ORB) technology that was adopted by the OMG. An ORB provides the mechanisms by which distributed objects transparently make requests and receive responses. The ORB enables object-oriented applications on different machines to communicate and interoperate.

The OMG has defined an Object Management Architecture object model. In this model, objects provide services, and clients issue requests for those services. The ORB facilitates this model by delivering requests to objects and returning any output values to the client. The services that the ORB provides are transparent to the client.

To request a service, a client needs the object reference for the object that is to provide the service. The ORB uses this object reference to identify and locate the object. The ORB activates the object if it is not already executing and directs the request to the object.

When test clients execute, they are instructed to run for a specified amount of time. They report failure and status information back to a central location. Upon completion of the system test, clients and servers are stopped, temporary files are removed, and final summary reports are produced.

The Test System

To manage all the activities of distributed system testing, we developed a test infrastructure that met our current needs and could evolve with new technologies and new needs. We followed a modular, object-oriented design approach to accomplish this.

We first engaged in several brainstorming sessions to produce a list of requirements for a complete distributed testing framework. This was an attempt to pinpoint all the attributes and functionality that a "perfect" test infrastructure would have, and it was done in the context of system testing distributed objects. The needs of product developers, test developers, and testers were considered, as well as the need to report metrics to the project team. The main focus, however, was on the two groups who would use the test framework the most: test developers and testers. Often these are the same people, but the distinction was made to clearly differentiate the needs of each group.

Product developers normally want quick and simple tests to verify that their code behaves correctly and at the same time have their programs work as they would for an end user. They don't want to be distracted by the infrastructure. Existing test APIs tend to be intrusive, requiring developers to have knowledge of the test environment in which their tests will be run. Therefore, we wanted our new test framework to minimize intrusiveness. This would allow developers to focus on testing the proper behavior of their code and not on the test infrastructure. Ideally, product developers should be able to write their tests with minimum restrictions, and the tests should plug and play in many different testing situations.

Test developers, whose job it is to develop ways to test the product, have many of the same needs as product developers but are more concerned with black-box testing and trying to "break" the product rather than verifying correct behavior. To do this, test developers want to be able to plug new tests into the test environment easily and quickly, and they want process and environment control. This would allow them to use the same tests in different scenarios to find more product defects. Test developers are usually the ones responsible for supporting the testing infrastructure. Thus, more than any of the other groups, test developers need a framework that is extensible, reusable, flexible, and controlled, and hopefully has a long lifetime. If a testing infrastructure becomes out of date, test developers will have to repair or replace it.

Often test developers are the ones who perform system testing, but many times this role is handed off to testers. Although the needs of both groups clearly overlap, testers need a testing infrastructure that is easy to use for the installation, configuration, and execution of tests. In many of our past projects, testing was done by temporary personnel. This freed

test developers to write more tests and assist product developers in debugging. When the test infrastructure is easy to use, the testing role can be handed off to testers earlier in the testing process. Additionally, the ability to reconfigure the test environment easily and quickly allows more scenarios to be tested. This increases the likelihood of finding more product defects, which leads to a better quality product.

Finally, test results are usually provided to the project team in the form of metrics. Gathering metrics in a distributed environment can be time-consuming. Data can be located on multiple systems on the network. However, when dealing with multiple processes running in parallel on different systems, results may not always occur in a consistent order. This implies the need for a centralized repository for testing results. This would make the generation of metrics much easier and faster, while providing a central location for finding problems and debugging.

Design Methodology

Taking into consideration the needs of the different groups mentioned above, we decided that the following attributes were required for our test infrastructure.

- Extensibility. Ensure the evolution of a modular system that can be dealt with on a component-by-component basis.
- Reusability. Allow object and code reuse for both tests and the test infrastructure.
- Flexibility. Provide a plug-and-play environment that allows for flexibility in test writing and configuration.
- Simulation. Provide the ability to simulate customer environments.
- Control. Provide centralized control of the test processes and environment.
- Nonintrusive. Hide as much of the testing infrastructure as possible from the system under test.
- Ease of use. Provide ease of use for installation, setup, configuration, execution, results gathering , and test distribution.

With these attributes in mind, we set about deciding on the basic set of classes that would be needed. We used a method for object-oriented design called Object Modeling Technique (OMT)[1] to develop a diagram showing class relationships (see "Object-Oriented Programming" on page 79).

We walked through several scenarios and expanded and refined our set of classes. Once we had an initial design we wrote CRC (class, responsibility, and collaboration)[2] cards for each of the classes in our design. (CRC cards are also described on page 79.) This design was reviewed by the product development team and their feedback was incorporated.

The Object Testing Framework

The design process produced an object-oriented software testing system that we named the object testing framework (OTF). Although this design is intended to test distributed object-based software, it can also be used to test distributed, procedurally based client/server software. The OTF consists of the classes shown in Fig. 2. The architecture of the OTF is such that there is a single master test control system (OTF management system in Fig. 2) that orchestrates running tests on multiple systems under test. This master system can also be a system under test.

In the following design discussion, the term object can mean class or an instance of a class. It should be clear from the context of the discussion which is meant.

OTF Management System

The OTF management system consists of the six major classes: user interface, OTF controller, test suite configuration, test controller, report generator, and database controller. This system provides the user interface that the software tester interacts with. Through this interface the tester specifies test configurations such as which client and server programs will be running on which SUTs. The OTF management system takes the specified configurations and makes them available to each of the SUTs, ensures that the SUTs run the specified tests, logs test data and results, and generates test reports.

The main class in the OTF management system is the OTF controller, which serves as the delegator object. It takes requests from the user interface object and manages the activities of the test suite configuration, test controller, and report generator objects. The test suite configuration object is actually created by the OTF controller. For a new configuration the object will initialize from the configuration data provided by the user interface. For a previously specified configuration, the object will initialize from the database. After this object's configuration data has been set, its primary responsibility is to respond to configuration queries from the SUTs.

The test controller has the overall responsibility for coordinating the running of tests on the SUTs. It provides the SUTs with a pointer to the test suite configuration object, synchronizes the starting of tests, and passes status data and requests back to the OTF controller. It also has the capability to log status data to the database via the database controller.

The report generator, upon a request from the OTF controller, queries the database controller to assemble, filter, and format test data into user-specified test reports. Raw test data is put into the database by each SUT's TestEnvironment object, while test process status data is put into the database by the test controller as mentioned above.

System under Test

Each system under test (SUT) contains fifteen classes. In normal operation, a SUT retrieves configuration data from the OTF management system, and then, based on that data, retrieves the specified tests from the management system. Since the SUT has the capability to build test executables from source code, it can retrieve test source code and executables from the OTF management system. Once the test executables are in place and any specified test setup has been completed, the SUT waits for a management system request to start the tests. When this happens, the SUT is responsible for running the tests, logging status, test data, and results, and cleaning up upon test completion.

The main object in the SUT is the host daemon, which is the SUT's delegator object. The host daemon takes requests from and forwards requests to the OTF management system and manages the activities of the setter upper, test executor, cleaner upper, process controller, and TestEnvironment objects.

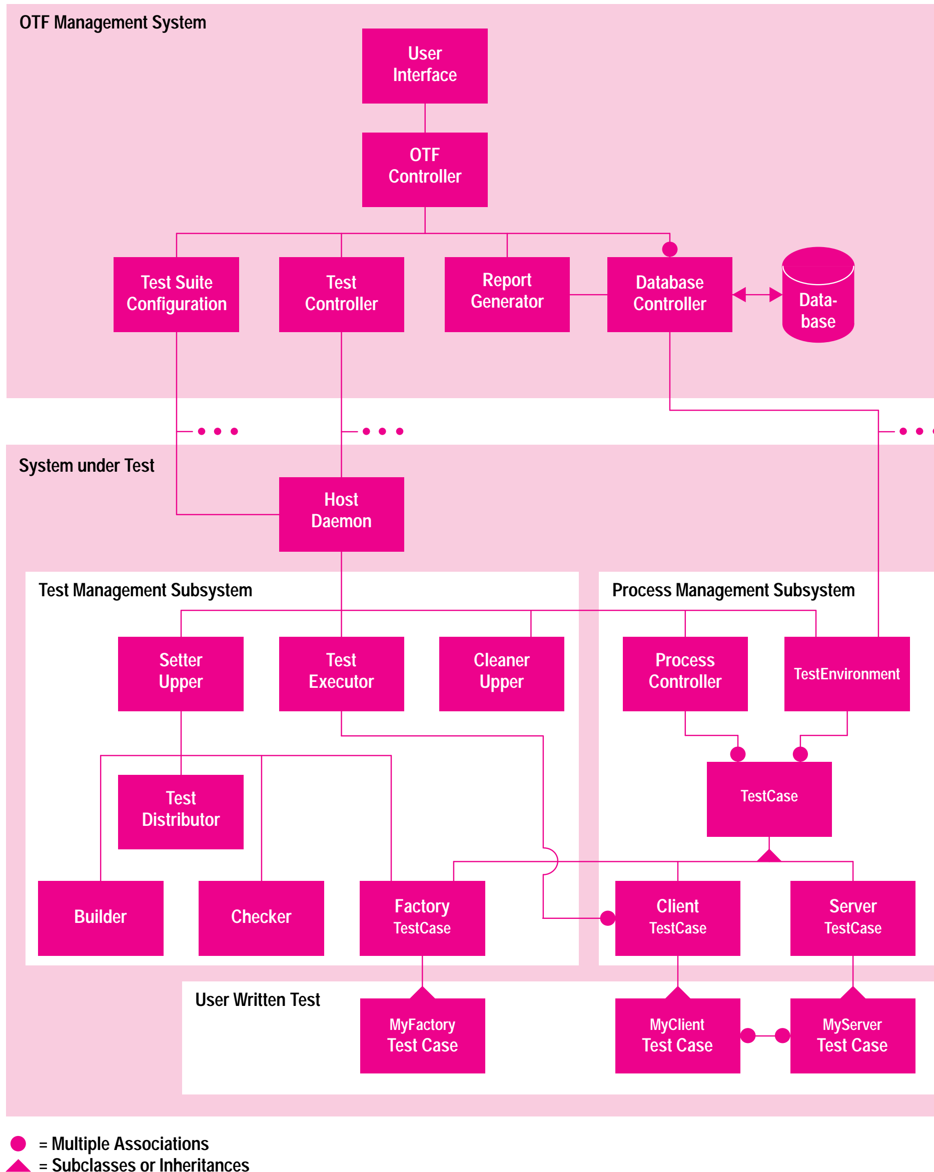

Fig. 2. System architecture for the object testing framework.

The overall responsibility of the setter upper, test executor, and cleaner upper objects is to manage how the tests are run. These three objects collaborate with the builder, test distributor, checker, and factory TestCase objects to form the test management subsystem shown in Fig. 2. The process controller and TestEnvironment objects provide the infrastructure for connecting the tests to the framework. These two objects collaborate with the TestCase objects to form the process management subsystem.

Test Management Subsystem

This subsystem sets up and executes the tests and then cleans up after the tests have completed. The setter upper is the object that controls test setup. It is a low-level delegator that manages the activities of the builder, test distributor, checker, and factory TestCase objects. The test distributor is responsible for retrieving test executables and sources from the OTF management system. When it retrieves source code, the builder is responsible for generating test executables from the code. How the tests are retrieved depends on the overall system environment and resources available. A distributed file system, like NFS, could be used, or the tests could be remote copied from the management system to the SUT. An important design consideration was to have a single repository for tests. This makes it easy to control changes to tests and is not intrusive on the SUTs.

The checker provides the ability to customize test setup by invoking a user-written program that can ensure that elements outside of the test environment are set up correctly. For example, it could check that NFS and DCE are running, that the display is set correctly, and so on.

The factory TestCase provides the setup procedures that arise when testing a CORBA-based distributed object system. It creates the CORBA objects that reside in the CORBA-based server TestCases and stores references to these objects for use by the client TestCases. The factory TestCase class inherits from the TestCase base class and the test developer writes a class that inherits from the factory TestCase class. This allows the test developer to customize the factory TestCase functionality for a specific test.

The test executor object starts the client TestCases through the functionality inherited from the TestCase base class. It also

Object-Oriented Programming

Object-oriented programming is a set of techniques for creating objects and assembling them into a system that provides a desired functionality. An object is a software model composed of data and operations. An object's operations are used to manipulate its data. Objects may model things such as queues, stacks, windows, or circuit components. When assembled into a system, the objects function as delegators, devices, or models. Messages are passed between the objects to produce the desired results.

The eventual goal of object-oriented programming is to reduce the cost of software development by creating reusable and maintainable code. This is achieved by using three features of object-oriented programming: encapsulation, polymorphism, and inheritance. Encapsulation consists of data abstraction and data hiding. Data abstraction is a process of isolating attributes or qualities of something into an object model. With data hiding an object may contain its own private data and methods, which it uses to perform its operations. By using polymorphism, an object's operation may respond to many different types of data (e.g., graphical and textual). Finally, using inheritance, an object can inherit attributes from other objects. The object may then only need to add the attributes, methods, and data structures that make it different from the object from which it inherited its basic characteristics.

For the design of the object testing framework described in the accompanying article, we used an object-oriented software design methodology called object modeling technique (OMT). This methodology provides a collection of techniques and notation for designing an object-oriented application.

One important aspect of object-oriented design, or any software design, is deciding on who (i.e., module or object) is responsible for doing what. A technique provided in OMT involves using an index card to represent object classes. These cards are called CRC (class, responsibility, and collaboration) cards. The information on one of these cards includes the name of the class being described, a description of the problem the class is supposed to solve, and a list of other classes that provide services needed by the class being defined.

reports back to the host daemon the success or failure of a test start.

The cleaner upper cleans up after the tests have completed. This may include removing temporary files, removing test executables, and so on.

Process Management Subsystem

The two main objects in the process management subsystem are the process controller and TestEnvironment objects. The process controller has the overall responsibility to monitor all test-related processes on the SUT. It can register or unregister processes, kill processes, and report process status back to the host daemon.

The TestEnvironment class provides the test developer with an application programming interface to the OTF. It provides methods for aborting tests, logging test data and results, checking for exceptions, getting environment variables, and so on. The test developer gets access to these methods through the base TestCase class, which has an association with the TestEnvironment class.

Creating a test involves writing a class that inherits from either the client TestCase or server TestCase base classes. The initialization and setup functionality for the test would be included in the test's constructor. The cleanup required when the test is done is included in the destructor. Finally, an implementation for the inherited run_body() method is included, which is the test executable that runs the test. The OTF API is made available through the pointer to the TestEnvironment class provided by the base TestCase class.

Implementation Approach

Once the design was complete, an initial investigation was made to find an existing system that matched the characteristics of the design. When no system was found, an analysis was done to determine the cost of implementing the new infrastructure.

It quickly became obvious that the transition to the new infrastructure would have to be gradual since we did not want to impact the HP ORB Plus product release cycle. The flexibility provided by an object-oriented system enables gradual migration and evolution through encapsulation, inheritance, and polymorphism. Tests could be isolated from the infrastructure so that new tests could be developed and evolved without modification as the infrastructure evolved. This flexibility fit nicely with the realization that the time to replace the existing infrastructure exceeded an average product life cycle.

Object-oriented encapsulation provided another advantage. Once some basic changes were made to the existing test infrastructure and tests had been converted to the new object-oriented programming model, the existing test infrastructure could be used to simulate some aspects of the new infrastructure. This allowed our system testing efforts to benefit immediately from the features of the new test system.

The development of the current version of the object testing framework has taken place in two steps, which have spanned three releases of the HP ORB Plus product. At each step we have continued to apply the same design principles. This work is summarized in the following sections.

First Step. For the first step, the goal was to consolidate the best practices of three existing test infrastructures into a single infrastructure that simulated as much of the major functionality of the OTF as possible. So as not to impact the ongoing HP ORB Plus software releases, another goal was to minimize changes to existing test code. This resulted in an infrastructure that consisted of a layer of shell scripts on top of two existing test harness tools. This significantly reduced the effort needed to set up, administer, and update the network of systems that were used to system test the HP ORB Plus product, while the tests continued to use existing APIs. It also confirmed that our design was indeed trying to solve the right problems.

Second Step. For this step the goal was to deploy the test developer's API to the OTF. The result was the implementation of the C++ TestEnvironment and TestCase classes described above.

Additional classes were designed to connect the TestEnvironment and TestCase classes to the existing infrastructure, but their existence is hidden from the test developer. This provides a stable API without limiting future enhancements to the infrastructure. Once the new infrastructure was deployed, we focused on porting existing tests to it.

This framework has resulted in minimal changes to existing tests and maximum increase in functionality for the tests. Most of the work simply involved taking existing code and

wrapping it in the appropriate class. All of our tests have benefited from the features provided by the TestEnvironment and TestCase classes and are insulated from changes to the framework.

At this stage of its development the object testing framework allowed the removal of intrusive test code required by the old test APIs. For example, many tests included code that allowed a test to be reexecuted for a specific time period. That code was removed from the tests because the same functionality is now provided by the framework.

In addition to supporting the design shown in Fig. 2, our current implementation provides the following functionality:

- Iteration. A test can be executed repetitively, either by specifying a number of iterations or the amount of time.
- Context-sensitive execution. The object testing framework behaves differently depending on how it is invoked. In the developer's environment it is transparent and does not affect test behavior. In the testing environment it is bound to the test system. For example, in the developer environment, the C++ functions cerr and cout go to the terminal, but in the testing environment they go to a file and the test report journal respectively. This encourages developers to put all existing tests into the framework because the test continues to work the way it did before it was ported.
- Simple naming service. A naming service allows the user to associate a symbolic name with a particular value such as the path name of a data file. In a distributed system, it is necessary for multiple processes to share values that are obtained outside of the system—for example, object references.
- Automatic capture of standard output streams cout and cerr. To simplify porting of existing tests, the cout and cerr streams are mapped to a file and the journal file respectively.
- Encapsulation of functions from the product under test. For example, the parts of CORBA used by the tests are encapsulated. As CORBA evolves and changes its C++ language bindings, only a single copy of the bindings in the framework has to change.
- Inheritance and reuse. Inheritance allows the test case developer to describe similar tests as a family of test cases derived from a common class (which in turn is derived from the TestCase class). In this case, polymorphism allows test code to be reused in multiple tests, while allowing changes to specific operations and data when needed.

Example. Our experience with the current framework has shown that the time to port existing applications tests to the new API is minimal. Fig. 3 shows an example of how test code would look before and after being ported to the new test infrastructure. This example is an implementation of the client for an OMG CORBA program, which simply prints "hello, world." In this case, the phrase will be printed by the say_it method provided by the server code. The following descriptions point out some of the differences between the two source files. The numbers associated with the descriptions correspond to the numbers in Fig. 3.

1. Fig. 3b shows portions of the test code with TestCase and TestEnvironment instrumentation. These classes are not in the code in Fig. 3a.

2. Fig. 3b includes the class declaration and method definitions of a HelloWorldTest class and a macro to register the definition with the TestEnvironment. The HelloWorldTest class is derived from the TestCase base class. Fig. 3a source has no HelloWorldTest class.

3. The check_ev_and_ptr macro in the Fig. 3a source is greatly simplified in the Fig. 3b source, thanks to the TestEnvironment's print_exception and is_nil methods.

4. Fig. 3a has a main function, whereas in Fig. 3b, the main function is replaced by the HelloWorldTest constructor, destructor, and run_body methods. This structure allows the OTF to instantiate and run the test code as needed. The constructor and destructor allow the test writer to separate out "execute once" code if desired. The run_body method may be executed more than once.

5. Fig. 3a uses a file to store the object reference string created by the server. (Use of files is potentially difficult if the client and server are on different machines.) Fig. 3b uses the TestEnvironment's naming service to get the object reference string.

6. In Fig. 3b argc and argv are not available as input parameters and must be obtained from the associated TestEnvironment.

7. The int return parameter of the main function in Fig. 3a is replaced by the TestEnvironment::Result of the run_body method in Fig. 3b. The effect is the same, to return the success or failure of the invocation.

Next Steps. The following is a list of items under consideration for implementing the rest of the design and adding more functionality to the TestEnvironment and TestCase classes.

- User interface class. We are investigating the possibility of encapsulating a graphical user interface that was designed for one of the existing test infrastructures.
- Test controller class. Here again we are looking at encapsulating an existing test synchronization controller.
- Memory leak detection. By adding this feature to the TestEnvironment and TestCase classes, all tests will get this functionality through inheritance.
- Integration with run-time debugging. This will improve tracing and fault isolation in a distributed, multithreaded environment.
- Heterogeneous networks. The current object testing framework handles networks of HP-UX* systems only. We need to expand the framework to handle other UNIX® systems as well as PC operating systems.

Summary

The object testing framework is based on using object-oriented technology to create a test infrastructure that is based on a number of small, self-contained modules and then developing these modules in a way that allows the testing effort to proceed while the test infrastructure continues to evolve. Each step of the evolution results in a usable test infrastructure that keeps the test effort online and provides critically needed support to product releases.

In addition to our overall commitment to complete this project, and a desire to see it used in other organizations in HP

```
// Standard C++ headers
#include <fstream.h>
#include <string.h>

// Header for CORBA HelloWorld object.
#include <helloTypes.hh>
// List of error messages.
extern char *msgs[];

// Simple macro to check for exceptions and valid pointers.
#define check_ev_and_ptr(ev, ptr, errcode)
// First, check for exception.
if ( ev.exception() ) {
}

cerr << msgs[errcode] << " Exception returned." << endl; \
        return errcode; \

// Next, check for valid pointer.

if ( CORBA::is_nil(ptr) ) {
cerr << msgs[errcode] << "Pointer is null." << endl; \
        return errcode;                                  \

int
main(int argc, char *argv[])
{

CORBA::Environment ev;

// Open a file which contains the object reference string for
// the Hello interface. Read the string.

ifstream f("hello_instance");
if ( !f ) {
cerr << "Could not open \"hello_instance\" file." << endl;
return 1;
}

}

char soref[1024];
f >> soref;
if ( !f ) {
cerr << "Could not read the stringified object reference"
    << "from the \"hello_instance\" file."
    << endl;
    return 2;

// Initialize the CORBA environement, get pointer to the ORB.

CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
CORBA::HPORBid, ev);
check_ev_and_ptr(ev, orb, 3);

// Convert object reference string to object reference.

CORBA::Object_var hello_objref
= orb–>string_to_object(soref, ev);
check_ev_and_ptr(ev, hello_objref, 4);

        •
        •
        •

CORBA::string_free(message);
CORBA::release(hello);
CORBA::release(hello_objref);
CORBA::release(orb);

}
```

(a)

```
// Standard C++ headers
#include <fstream.h>
#include <string.h>
// Header for CORBA HelloWorld object.
#include <helloTypes.hh>

// Header for Test Case object.
#include "testcase.hh"

// List of error messages.
extern char *msgs[];

// Simple macro to check for exceptions and valid pointers.
#define check_ev_and ptr(ev, ptr, errcode)

// Check for exception and valid pointer.

    if ( te.print_exception(ev) || te.is_nil(ptr))
      return (TestEnvironment::Result)
        (TestEnvironment::user_code + errcode);

// Declaration of HelloWorldTest class.
class HelloWorldTest : public TestCase
{
public:
 HelloWorldTest();
 ~HelloWorldTest();
 TestEnvironment::Result run_body();
};
private:
 CORBA::ORB_ptr orb;
 HelloWorld_ptr hello;
 char *message;

DECLARE_TESTCASE_FACTORY(HelloWorldTest);

// Definition of HelloWorldTest constructor.
// The following pieces of the test are considered set up.
HelloWorldTest::HelloWorldTest()
{
CORBA::Environment ev;

// Get the object reference string for
// the Hello interface from the test environment.
string soref = te.get_object_string("hello_instance");

// Initialize the CORBA environment, get pointer to ORB
orb = CORBA::ORB_init(te.argc, te.argv, CORBA::HPORBid, ev);
check_ev and_ptr(ev, orb, 3);

// Convert object reference string to object reference.
CORBA::Object_var hello_objref
        = orb–>string_to_object(soref.c_str(), ev);
check_ev_and_ptr(ev, hello_objref, 4);

// Narrow the COBRA:: Object object reference to a HelloWorld one.
hello = HelloWorld:: _narrow(hello_objref, ev);
check_ev and ptr(ev, hello, 5);
COBRA:: release(hello_objref);

        •
        •
        •

CORBA::string_free(message);
CORBA::release(hello);
CORBA::release(orb);

}
```

(b)

Fig. 3. (a) Test code before being ported to the object testing framework. (b) The same test after being ported.

involved in distributed object technology, we will continue to participate in standards organizations such as OMG and X/Open to follow the work that is being done in the area of testing. To date, we have evaluated and provided feedback to the X/Open Consortium and have a representative monitoring the activity at OMG.

References

1. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

2. K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *SIGPLAN Notices,* Vol. 24, no. 10, October 1989.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

Open Software Foundation is a trademark of the Open Software Foundation in the U.S. and other countries.