

A Test Execution Environment Running Abstract Tests for Distributed Software

A. Hartman, A. Kirshin, K. Nagin

IBM Haifa Research Laboratory
Haifa University 31905
Haifa, ISRAEL
+972-4-8296277

{hartman,kirshin,nagin}
@il.ibm.com

ABSTRACT

This paper addresses the problem of executing abstract tests on distributed software. Although our focus is on specification based behavior testing that exploits model based test generation, we provide an environment that supports both automatic and manual test generation. Tests derived from a specification are by nature abstract. The tester must manually convert the abstract tests into their concrete executable form. We claim that a test execution environment designed to execute abstract tests can transform and execute these tests efficiently to verify distributed software. We also claim that the same abstract tests can be reused and extended to verify issues related to concurrency, stress, and input data variations. We describe the test execution environment and discuss an industrial trial that exploited a prototype of the system to improve the testing of a distributed application. The trial reused a behavior test suite generated for its function test and executed it in its system test. The reuse required no additional coding and was successful in discovering five additional defects related to concurrency.

Keywords

Distributed software testing, test execution environment, abstract test transformation and reuse, model based test generation and execution, concurrency testing, stress testing, input variation testing

1. INTRODUCTION

In recent years, software modeling has enjoyed tremendous popularity due to the widespread adoption of object-oriented models as an aid to software design [4]. The use of software models for the generation of abstract tests has also been reported in both academic settings ([1], [5] and [11]), and in practical experiments ([3], [6] and [13]). However, the adoption of a specification based modeling strategy for generating test suites has yet to gain industrial momentum. One of the impediments to its adoption is that most current test execution environments were not designed to run the generated tests.

Our research deals with a testing architecture to validate the behavior of distributed programs. The distributed application under test is not restricted to any particular programming language or operating platform. An abstract test case describes test logic without concerning itself with platform or language issues. In order to run the test, the execution environment must deal with the messy details of concrete execution. Furthermore, the abstract tests designed to validate basic functional behavior may be reused and extended to verify additional software requirements such as concurrency.

1.1 System Under Test

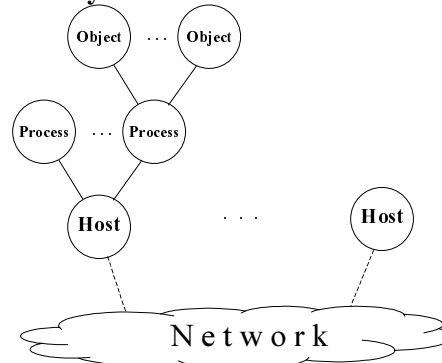


Figure 1 Abstract model of system under test

Figure 1 shows an abstract model of a system under test (SUT). An SUT may be perceived of as a set of objects. The abstract test describes the objects' expected behavior, state, and output in response to a sequence of stimuli from the tester.

The concrete implementation of the SUT runs within different process address spaces; the processes reside within hosts interconnected by a network.

1.2 Test Execution Environment

Our test execution environment is an infrastructure that supports the testing of distributed software. Software testing environments are applications that assist testers in executing tests and collecting results against the application under test. Testing environments usually parallel the

applications they test. They run under the same operating system and use the same programming languages to interact with the SUT. They also provide services to support interfaces and libraries for testing scripts. The cost of creating and supporting these testing environment is a significant part of the overall development cost.

The cost and complexity of the testing environment increases when the software under test is distributed. Distributed software may run in different host environments, separated by a network. Different programming languages may be used to code the different components in the software under test.

For example, a server may run under Linux and its clients run under Windows. The server and clients communicate over a TCP/IP network. The server may be coded in Java and the clients in C++.

The testing environment must provide testing solutions for all of these heterogeneous application environments and languages. The creation and maintenance of complex testing environments is a distraction for a test team. Their main focus is on validating the software under test.

The primary goal of our test execution environment is to make software testing more efficient by providing an infrastructure to support these heterogeneous application environments and languages. Of course, the infrastructure must scale down to support more simple single language non-distributed software testing.

1.3 Abstract Test Execution

An abstract test for behavior testing is a scenario involving communication between the tester and the system under test. It specifies a sequence of tester stimuli to the SUT, observation of SUT responses, and the assignment of a verdict based on the observations. An abstract test is designed to exercise a particular execution sequence or verify compliance with a specific requirement. The word “abstract” refers to the fact that the test case is platform and programming language independent. Examples of abstract test formats are TTCN-3 [14] and the AGEDIS [2] abstract test.

The test execution environment is responsible for the concretization of the abstract test. It must implement the test logic on a set of operating platforms using a set of different programming languages. If the test scenario occurs on multiple hosts, it is required to distribute the test stimuli and observations over a network.

Abstract tests have the advantage that they express the testing logic without getting bogged down in the details of how to execute the test. Executable tests are written in the programming language of the system under test or in a test scripting language. Thus, the abstract test must be translated in order to execute it on the SUT. The test execution engine

bridges the gap between the abstract test and the SUT by mapping the abstract stimuli to method invocations, mouse clicks, or system commands, and the abstract observations to actual value checking. The test execution environment executes the translated test directly and checks the observed results against the expected results as predicted by the model or as described in the abstract test.

The abstract test is a required input to the test execution environment, telling it **what** to execute against the SUT. Test execution directives are another input to the environment, and provide the information on **how** to execute the abstract test. These directives include information about target platforms, programming languages, actual input data, data types, and other mappings needed to make the abstract test concrete.

1.4 Outline

Section 2 describes the test execution environment design. The design provides optimal support for executing abstract tests for distributed applications.

Section 3 describes how we extend the abstract tests with object partitioning to cover testing issues related to concurrency, stress, and input variations.

Section 4 describes an industrial trial that exploited a prototype of the ideas presented in Sections 2 and 3 to execute a test generated from a model. The test was run in function test and then reused in system test with no additional cost.

2. Test Execution Environment Design

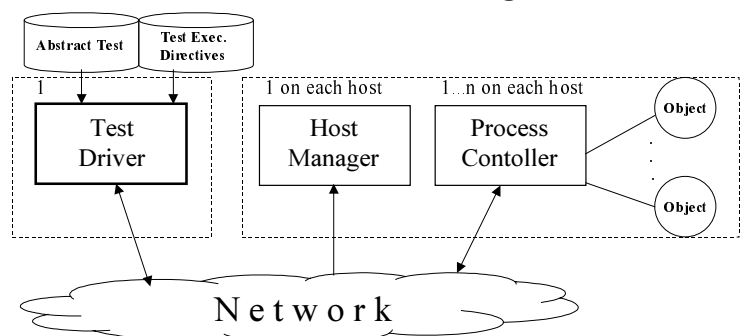


Figure 2 Test Execution Environment Design

Our test execution environment design supports the basic functionality for testing distributed systems: starting processes, creating SUT objects on remote hosts, controlling and observing them, and logging a trace of the test execution.

Figure 2 illustrates the main components of the test execution environment. The *test driver* manages the test execution and communicates with the SUT via the *host managers* and *process controllers*. The host managers are replicated on each target host. The processor controllers

interact directly with the SUT objects. We describe the components in greater detail below.

The test driver and SUT may run on the same host or on different hosts, and they need not be coded in the same programming language. The driver communicates with host managers and process controllers in order to accomplish the following tasks:

- Create, destroy, and locate process controllers (via host manager).
- Create and destroy SUT objects (via process controllers).
- Control input to the SUT objects (via process controllers). Control means invoking procedures and functions, and sending input messages.
- Observe output and state of the SUT objects (via process controllers). Observation means querying an object's state, recording function responses, publishing asynchronous messages, and getting an output.

Only the test driver component understands the syntax and semantics of the abstract test cases. The other components handle real world interactions. This separation of concerns makes the test execution environment design well-suited for the testing of distributed software.

The test driver is by far the most complicated component in the test execution environment. It is implemented only once since it has no SUT platform or language dependencies. On the other hand the host manager and process controller designs are simpler, but their implementation is dependent on the target SUT. Host managers must be implemented in all target SUT platforms. A process controller runs in the same process as the SUT object, and it simplifies interfacing with the SUT when the process controller and SUT are in the same programming language. Thus the process controller must be implemented in all target SUT platforms and programming languages.

3. Abstract Tests Extended with Object Partitioning

The SUT consists of different entities whose behavior is described in the abstract test. The tester can partition the abstract test into a set of executable objects. For instance, a client/server application can be treated as a set of client objects and a server object. A file system can be treated as a set of user objects, a set of file objects, and a file server. The test modeling language does not have to be object oriented and neither does the abstract test. However, it is required that the abstract stimuli and observable responses be identifiable so they can be mapped to concrete objects using the test execution directives.

For example, consider an SUT that is a client/server application. The application server connects its clients to a

database of accounts. Unique keys index the database accounts. The client can create, remove, or update accounts based on the keys. An administrator can start or stop the server. The abstract test contains different interleaving of these method calls. We illustrate a simple test case below:

1. Start the application server.
2. Observe that no exception occurred.
3. Create an account with key1.
4. Observe that no exception occurred.
5. Create another account with key1.
6. Observe that an exception is thrown.
7. Stop the application server.
8. Observe that no exception occurred.
9. If all the above observations are as expected, the test case passes; otherwise, it fails.

The test can be partitioned into a client object and an application server object in the test execution directives. The create(), remove(), and update() methods are mapped to a client object and the start() and stop() methods are mapped to an application server object. When executing the test, our execution environment does the following:

- Creates both the application server and client object.
- Calls the client object's methods create(), remove(), and update(). The client object, in turn, communicates with the application server object, which, in turn, communicates with the database.
- Calls the application server object's methods start() and stop().
- Compares the responses from both the application server and client objects with the expected responses.

In the following sections we show how our test execution engine can exploit this object partitioning to extend the test usage.

3.1 Object Distribution

When the SUT is a distributed application, the test runs on multiple hosts. The test execution environment simplifies the distribution and control of a test case. The test driver causes the host managers to create process controllers and the process controllers create the SUT objects. The test driver then coordinates all interaction between or within the test objects, based on the abstract test. This includes object initialization and cloning, which is discussed below.

In the above example, the client, application server, and the test driver can all reside on different hosts.

3.2 Object Multiplication and Cloning

The abstract test may be re-used for concurrency and stress testing by multiplying the SUT objects and the environment in which they execute.

The abstract test is executed in a stepwise fashion. The test driver synchronizes the test execution to run multiple independent operations concurrently or sequentially. At each step, it runs the required number of independent operations and waits for their completion before checking the results and continuing to the next test step. If the actual results are different from the expected results, the exact sequence that triggered the fault is known. The sequence can be replayed when a fix is provided for the fault. The test driver also allows the user to step through a test case interactively when debugging a fault.

3.2.1 Mutual-independence of Objects

When the number of objects does not affect the test behavior, these objects are said to be *mutually independent*.

Sometimes the tester can identify some shared attribute that distinguishes the SUT objects, but allows the test behavior to remain invariant when many objects are instantiated. In this case, the test execution environment can reuse the simple abstract test by running each test step against many SUT objects in parallel. We refer to the creation of many instances of the same SUT object as *cloning*. Clones are created alike but they are made distinct by initializing each clone with different values.

In the client/server system there may be many clients and servers referring to the same database. The testing model describes a single client's interaction with a single application server. It is still necessary to verify that the SUT works properly in the presence of many clients and servers. Object mutual-independence is achieved by initializing each client clone with a different set of primary keys used to access the database.

Cloning may take place at the object level, the process level, or even at the host level of the hierarchy illustrated in Figure 1.

Object multiplication and cloning is an efficient way to validate concurrency issues and stress the SUT since it re-uses an existing abstract test. Once the test has run successfully without object multiplication, any problems discovered subsequently can be attributed to the introduction of more objects into the system. The test driver can reproduce the precise synchronization scenario that introduced the fault, thereby facilitating the fixing of the bug.

3.2.2 Object Synchronization

The test driver implements two types of synchronization of cloned objects: *concurrent* and *sequential*.

Concurrent synchronization causes each abstract stimulus and observation specified in the abstract test to start on all clones at the same time. The next stimulus and observation does not start until the previous one is completed on all clones.

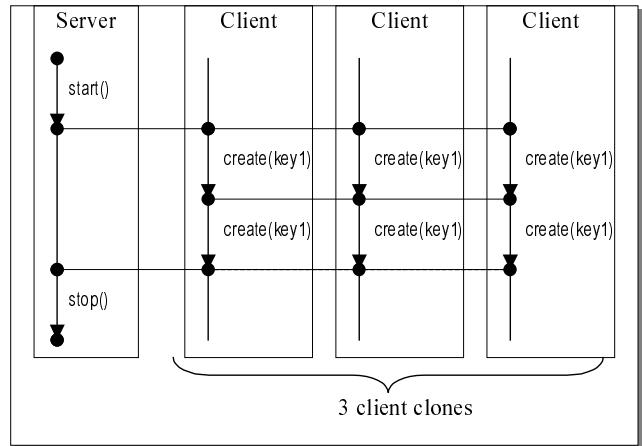


Figure 3 Concurrent Synchronization

Figure 3 illustrates concurrent synchronization, in which the client/server abstract test is extended to run with three client clones and one application server. Each create() method invocation is repeated concurrently on all of the client clones. The test driver waits until all the create() methods return before proceeding to the next step in the test.

Sequential synchronization causes each abstract stimulus and observation to start and complete on each clone, before proceeding to the next one.

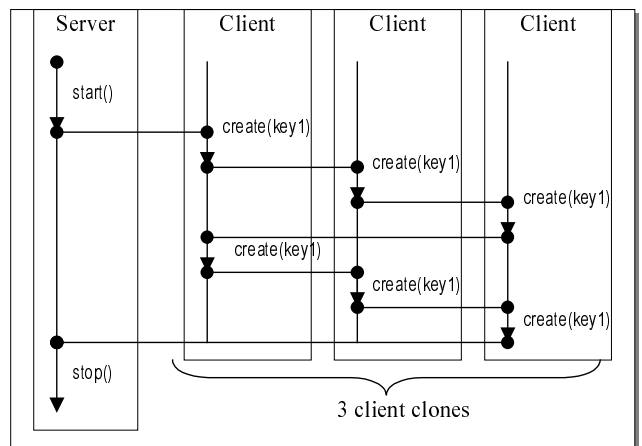


Figure 4 Sequential Synchronization

Figure 4 illustrates sequential synchronization. Each create() method invocation is repeated on each of the clones in sequence. The test driver waits until each create() method returns before proceeding to the next step in the test.

3.3 Object Initialization

Object initialization is used to create mutual-independence between objects. The test driver creates process controllers and objects dynamically during testing. It uses a hierarchical structure to organize them (see Figure 1). Objects are the children of process controllers and process controllers are the children of host managers.

Each host manager, process controller, or object is initialized with initialization sets containing name-value pairs specified in the test directives. Value pairs may reference values belonging to other members that are further up the hierarchy.

For example: If a host initialization set includes: **IP = 9.148.32.112**, and one of its processes has an initialization set which includes: **USER = Joe**, then any object of the process may include: **IP = HOST.IP; USER = PROCESS.USER** in its initialization set.

The initialization set of an object may also reference the following poly-morphic functions, which return objects of the appropriate class:

- **unique()** – returns a unique object
- **min()** – returns the minimum object in the host set
- **max()** – returns the largest object in the host set

An initialization set can have an enumerated range of values. This is used to initialize clones differently. Values are assigned to the clones cyclically in sequence.

For example, we initialize the client/server account example with four client clones using the following initialization set:

```
key1 = unique()
amount = 0, min(), max()
name = "Harry Potter"
```

Four clients are instantiated, each is assigned a unique key1, the name of each clone is "Harry Potter", and two client amounts are 0, one is min() and one is max().

3.4 Objects and Input Variation testing

The abstract test may also be re-used to do input variation testing [6](i.e., running the same test scenario with different input data). The data initialization scheme uses a data generation coverage strategy similar to that described by Cohen *et. al.* [8].

4. Industrial Experience

GOTCHA-TCBeans [10] is an IBM internal test automation toolset developed at the IBM Research Laboratory in Haifa.. GOTCHA is a modeling tool used to generate abstract tests. It is based on the Murφ [9][8] description language. Murφ is not an object oriented programming language. TCBeans is a prototype of our test execution environment. GOTCHA-TCBeans has been used successfully to test IBM applications for the past four years. One example of its use is described in detail below.

4.1 Customer Support Center

The SUT was the dispatcher of an electronic customer support center. The customer support center is a component based distributed application written in Java.

Customers of the support center contact agents and request support. The agents are represented by software components called access points. Each access point generates interaction objects to represent its communications. Interactions can include multiple access points. The controllable stimuli and observable responses correspond to access point methods and variables. A dispatcher schedules interactions to be serviced by access points. The dispatcher has a complex scheduling policy based on the skills required to deal with the communication, and the skills possessed by the available agents.

We used GOTCHA to generate a set of abstract tests. The testing model included a few access points, interactions, and a dispatcher.

An independent testing group used TCBeans to execute the test in its function test. They discovered 35 defects; most of the defects were attributed to sequencing problems.

The test generation model encountered state explosion problems when only a few access points were modeled, so the model had to be constrained accordingly. However, testing with many access points was a requirement in its system test to validate the dispatcher under stress. The test team reused the abstract test but created many access points using object distribution, multiplication, and cloning in its system test. Without any additional coding, they discovered an additional five defects caused by running the test with concurrent stepwise synchronization.

5. Future Work

We would like to go beyond the stepwise concurrency scheme described above and run more complex synchronization patterns generated from an abstract model. For example, we could synchronize the execution of more than one instance of the same or different abstract tests and generate synchronization patterns that describe when to synchronize the stimuli and observations between the multiple test instances.

6. RELATED WORK

The AGEDIS [2] consortium is implementing the proposed test execution environment. The AGEDIS abstract test format is in the public domain. The AGEDIS test generator, TGV [12], GOTCHA and UCBT[15] all intend to write their test suites in the format. Thus testers may choose from a variety of modeling languages when creating their tests and still execute them in the same testing environment.

7. CONCLUSIONS

We described a test execution environment designed to execute abstract tests on distributed software. We described

how the environment transforms and executes the abstract tests to verify the distributed software. We also demonstrated that the same abstract tests may be reused and extended to verify issues related to concurrency, stress, and input data domains. Finally, we provided supporting evidence in the form of an industrial case study that our execution environment does indeed support testing for distributed application using abstract tests that have no notion of the software's distributed design. Most importantly, the trial reused the behavior test suite generated for its function test and executed it in its system test. The reuse required no additional coding but it discovered five additional defects related to concurrency.

8. ACKNOWLEDGMENTS

We would like to express our thanks to Gary Bosko for his help in running the industrial experiment.

9. REFERENCES

- [1] Abdurazik, A. and J. Offutt, J., Generating Tests from UML Specifications, Second International Conference on the Unified Modeling Language (UML99), 1999.
- [2] AGEDIS Consortium, Automated Generation and Execution of Test Suites for Distributed Component-based Software, <http://www.agedis.de/>
- [3] Apfelbaum, L. and Doyle, J., Model-based Testing, Proceedings of the 10th International Software Quality Week, QW97, May 1997.
- [4] Booch, G., Object Oriented Analysis and Design with Applications. Benjamin/Cummings, 2nd edition, 1994.
- [5] Carver, R.H., and Tai K.-C., Use of sequencing constraints for specification-based testing of concurrent programs. IEEE Transactions on Software Engineering, 24 (June 1998), 471-490.
- [6] Chaar, J. K., Halliday, M. J., Bhandari I.S., and Chillarege, "In-Process Evaluation for Software Inspection and Test", IEEE transactions on Software Engineering 19, 1055-1069 (November 1993)
- [7] Clarke, J. M, Automated Test Generation From a Behavioral Model, Proceedings of the 11th International Software Quality Week, QW98, May 1998.
- [8] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, The AETG System: An approach to Testing Based on Combinatorial Design. IEEE Transactions on Software Engineering, 23(7) 437-444, (July 1997).
- [9] Dill, D., Murø Description Language and Verifier, <http://sprout.stanford.edu/dill/murphi.html>.
- [10] Hartman, A., and Nagin, K.M., GOTCHA-TCBeans Tool Overview, Release 3.0.2, 2001 <http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html>.
- [11] Hartmann, J. , Imoberdorf C., and Meisinger M., "UML-based Integration Testing" in Proceedings of ISSA 2000.
- [12] Jard , C., Jeron, T., Jezequel ,J., Le_Traon Y., Pickin, S. Test Synthesis from UML Models of Distributed Software.. IEEE Transactions on Software Engineering, 24 (June 1998), 471-490.
- [13] Poston , R. M., Automated Testing From Object Models, Communications of the ACM, September 1994, 48-58.
- [14] TTCN-3. Draft new Recommendation ITU-T Z.140: The Tree and Tabular Combined Notation version 3 (Core Language), 2001
- [15] Williams, C., UCBT Used Case Based Testing, <http://www.research.ibm.com/softeng/TESTING/ucbt.htm>