Automating a Testing Environment

Ascert's software testing solutions make it possible to automate some of the most time-consuming and error-prone tasks associated with application testing. This article contains a high-level discussion of black-box, functional, and regression testing.

Black-Box Testing

Black-box testing is the type of testing normally performed by end users and QA groups. In this type of testing, system inputs and outputs are verified, but the internal processing of the application is a "black-box" to the tester. Black-box testing is usually carried out through the execution of a set of test cases where the contents of incoming requests are defined, as are the specific responses to the requests and the expected changes to the application database. Once defined, the test will be run, the results checked, and the test either signed off, or if the results are not as anticipated, a problem report will be created, the application corrected, and the test executed again. This cycle will be repeated until the expected results are obtained. Black-box tests of this type can be used to perform various types of tests, including functional and regression tests.

Functional and Regression Testing

Functional and regression testing are two of the most commonly performed types of testing. Functional testing consists of performing a test case or group of test cases to determine if a specific function of an application performs correctly. Regression tests are simply a collection of functional test cases that have been previously run and validated against an application. Sophisticated applications are rarely composed of a single component. Although the previous description of functional and regression testing refers to "application testing," both techniques may be applied at various application "levels," such as testing a single program, a subsystem consisting of multiple programs, or the entire application.

Functional Testing

Functional tests are performed to verify that a new application performs as expected and to verify specific changes or additions to an application's functionality.

Regression Testing

Regression testing is the process of fully exercising and verifying the functionality of an application. Regression tests are generally performed when an application has been modified, with the intent of verifying that the modifications have not caused any unexpected functional changes in the rest of the application. It may take hundreds or even thousands of test cases to perform a full regression test. The regression test may also need to be repeated several times for a single change to the application if errors are discovered during the test, and many times during the application's lifetime. Often for "minor" changes, regression testing is limited to the area of the application that has been changed rather than the complete functionality of the application as would be advisable. This is generally due to a lack of time rather than a lack of will to carry out the testing. This is where automating testing to the point where executing a full regression test is not a major exercise really begins to pay back.

Testing Automation

Application testing can be a very expensive, time-consuming process, especially if it is done manually. If we consider performing an annual regression test where only 100 test cases were defined, and each test case took only three minutes to execute and validate, each regression test would take 300 minutes. When you add the time it takes to set up the supporting test data, assuming one database entry per test case with each taking 30 seconds, the time for each run would be 350 minutes. If even a single problem is found during the regression test, the process will have to be repeated

bringing the total testing time to 700 minutes. Given this example for a small regression test of a simple application, it is easy to see why manually testing sophisticated applications is so expensive and time-consuming. Reducing the time and cost of testing is one of the primary goals of testing automation.

Testing automation is simply the process of using software tools to perform testing procedures that were previously done manually. Automation may be applied to some or all of the procedures involved in performing a test. The process of adding automation to any testing environment requires a certain amount of work to be done to implement the tools chosen to perform the automated tasks. Unfortunately, in many testing environments, the personnel responsible for performing the tests are already so busy that they have little time left to dedicate to implementing automation

This simple fact leads directly to the answer to the question: “Where do I start in the process of automating my environment?”

The answer is: “Identify your most time-consuming testing procedures and how much time is spent on each of them during a normal test; then estimate how long each would take to automate. The procedure with the highest ratio of time-consumed to time-to-automate is the procedure to automate first." In the example given in the paragraph above, this procedure might result in the following table:

Task	Time-consumed	Time-to-automate	Ratio
Execute Tests	150 min.	150 min.	1:1
Validate Tests	150 min.	300 min.	1:2
Set up Test Data	50 min.	10 min.	5:1

Looking at the table, it is obvious that although the process of setting up test data is not the biggest time consumer, it provides the most "bang for the buck" in terms of the benefit derived from automation. By automating this task first, 50 minutes of time is saved on each test run, which can then be dedicated to automating the remaining tasks on the list. One thing to note about the "Validate Tests" entry on our worksheet is that although it shows a Ratio of 1 :2, meaning that it will take twice as long to automate the task as it does to perform it, this does not mean that the task shouldn't be automated. Assuming that this task is executed during every test run, three test runs would result in a total time savings of 450 minutes and a net savings of 150 minutes. After 10 test runs, the total net benefit would be 1,200 minutes which far exceeds the net benefit of 490 minutes provided by automating the test data setup, for the same number of test runs.

Please note that the above calculations assume that after a task is automated, the amount of the tester's time consumed by the task will be zero. This is probably never going to be true in the real world, but remember these calculations are intended as a guide to where you should start your automation process and how you can create free time to continue it. You may wish to refine this approach for use in your environment. The previous descriptions of the uses of automated testing and the benefits it can provide are intended only as an example of some of the possibilities available. The benefits provided in any particular testing environment depend on the characteristics of that environment and the tools chosen for use in automating test procedures.

Automating the Testing

The tablet above tells us that the first procedure to automate is the set up of the test data. The most straightforward method would be to simply save a copy of the database after the data has been inserted manually and then restore this database prior to each test run. This automation could be performed using anyone of several *NonStop Kernel* utilities, such as Backup and Restore or by using a tool such as Ascert’s Relate to create a starting position in an existing database.

The procedure with the next-best ratio is "Execute Tests." In order to use a tool to automate this process, the first thing we need to do is take an analytical look at our example test environment and test case. Presuming that we are black-box testing, it is very likely that the source of test transactions is outside our application. Only the messages coming from or going to it are actually included within the application boundary, it is sometimes possible to use a physical device to send and receive transaction messages to /from the system. This may be appropriate for manual testing since the transaction has to be created someplace, but how the message is created is generally irrelevant from the point of view of the application, so long as the message is in the correct format and is delivered in the manner expected by the application

process. Removing the specification of an external device as the source/destination of external messages makes the test case more generic and it can be used in both manual and automated test scenarios. We can now just deliver a message containing the required information for our test transaction, and examine the response to see whether the transaction was handled correctly.

Automating Test Actions

Now that the test case clearly specifies the testing requirements, the test case action can be automated by creating a script that builds a "transaction-request" message containing the appropriate data and sends it to the application. As an example, VTALK is the VersaTest scripting language which is used to implement intelligent message processing. Scripts are then run using an execution engine; in VersaTest a process known as VPRO, runs compiled object of VTALK scripts. Once VersaTest is configured into the application, the script to create and send the required message and to receive and log the reply can be run to automate the test case. Once a single message and reply has been developed it is then a matter of adding other message formats and varying the data in those messages to automate test cases to test the full functionality of the application in a fully functional and reusable test suite.

Automating Test Validation

In order to perform validation, either manually or automatically, it is necessary that the expected results of a test be specified. For manual validation, the tester must use the expected results specified in the test case to compare against the actual results of the test. When automating the process of validation, the specifications of the test case generally are not accessible to the automation-tool; so another method of specifying the expected results is required. How might this be automated?

Transaction-Reply Validation

Automatically validating the message returned by the application may be challenging. Although the user can easily read a message displayed at a device and verify that it contains the expected text, a validation tool either has to intercept the message "in-flight" as it is being sent out of the application, or capture the message display at the device and compare it to a specified result. Retrieving application data from an external device display is possible with some tools. However, this method of validation expands the scope of our testing outside the application by involving the external device. If the device displays the application reply incorrectly because of problems with its internal processing, or even because an emulation window has been repositioned or resized, it may cause the test tool to report a problem with the application, when the real problem is within the device.

Because external validation of messages can be problematic, the preferred solution is to intercept the messages in-flight as they are sent out of the application. Very few products offer this functionality and many simulator programs have been written specifically to provide it. Unfortunately, most of these programs were written to support only a single device or interface, and they are often poorly documented and maintained. VersaTest provides this functionality in a robust, generic fashion that can be customized to support virtually any device or application interface.

Although performing validation in-flight within a script can be very useful in some testing situations, VersaTest provides a second, even more powerful, way of validating messages. This method depends on capturing the messages associated with a test run in a file and then automatically comparing these captured messages against a previously captured set of messages for the same set of transactions. The previously captured messages constitute the expected results data used for validation of the messages in the current test run.

This type of test validation is based on one of the most basic tenants of testing: that the same test run in the same environment will produce identical results every time it is run. This fundamental view is valid for our example application, but in real-world application testing, it becomes very difficult to execute tests in EXACTLY the same environment because of things like timestamps that are embedded in application messages and databases. Since it would be virtually impossible to guarantee that timestamps within transaction messages would be identical in two test runs, even if the system time were manipulated to be the same at the beginning of the test runs, it is important that factors such as the time or sequences and specific orders of events do not throw out the automatic checking of results. In order to do this, the VTALK scripting language allows specific message fields to be marked for exclusion during comparisons.

Database Validation

Verifying that the messages in and out of the black-box are valid is half the picture. Typically, the application itself will have

a database which is modified as transactions take place. In order to verify that the contents of the application database contain the correct values after a test run is executed, a set of expected results must be defined. These expected results could be specified in any number of ways. For manual testing our test case defines the expected results, but in order to implement automatic validation, the expected results have to be specified in a way that is accessible to the automation tool performing the validation. A data file containing multiple records, with each record containing fields that specify the name of the file to be examined and the values that should be contained in each record, would be one method for providing this information. The automation tool could then read this file and examine the file/record-value combinations it defines. If the combinations specified in the expected results file are found in the application's files, then the validation is considered complete and the test successful. Unfortunately, this simplistic approach leaves many potential problem areas unexamined. What happens if the application unexpectedly inserts a new record in one of the files? Or if the person in charge of maintaining the expected results file forgot to add the validation values for a new group of accounts that had been added to one of the files? Both of these scenarios would result in the validation tool incorrectly reporting that the database was correct.

Ascert's Relate product is designed to compare the contents of one file or table against the contents of another with the same, or at least similar, structure. Relate can be used to perform automatic validation of test databases by comparing the values stored in the application's files or tables after a test run to a second set of files or tables that contain the expected results. This is similar to the database validation approach described above, but the files or tables are used to contain the validation data and Relate obey files are used to specify the comparisons to be performed.

Relate is also very sophisticated in its comparison capabilities and can identify differences that can't be specified in an expected results file, such as extra rows in a table. As with the VersaTest message file comparison, this type of testing is based on the basic testing premise that if the same test is run in the same environment, it should produce the same result. This premise applies not only to messages passing through the system, but also to the database that is modified by the system.

Summary

We have shown how test tools like Ascert's VersaTest and Relate can be used to automate two of the most important and time-consuming areas of functional and regression type testing: test execution and test validation. This article did not discuss many of the potential uses of automated test tools, such as stress and volume testing and only touched on the sophisticated comparison capabilities of Relate. Every application-testing environment is different and therefore requires a unique solution. VersaTest and Relate have been designed to provide the flexibility to address virtually all *NonStop server-based* testing requirements and provide significant savings in both cost and testing time. By using tools to automate your testing environment, you will be able to test faster and cheaper, but most importantly, better.

This article was originally published in "The Connection", May/June 2002, Volume 23, No. 3