Black Book

ixia

Edition 10

Test Automation

http://www.ixiacom.com/blackbook

August 2010

Your feedback is welcome

Our goal in the preparation of this Black Book was to create high-value, high-quality content. Your feedback is an important ingredient that will help guide our future books.

If you have any comments regarding how we could improve the quality of this book, or suggestions for topics to be included in future Black Books, contact us at ProductMgmtBooklets@ixiacom.com.

Your feedback is greatly appreciated!

Contents

How to Read this Book

The book is structured as several standalone sections that discuss test methodologies by type. Every section starts by introducing the reader to relevant information from a technology and testing perspective.

Each test case has the following organization structure:

Overview	Provides background information specific to the test case.
Objective	Describes the goal of the test.
Setup	An illustration of the test configuration highlighting the test ports, simulated elements and other details.
Step-by-Step Instructions	Detailed configuration procedures using Ixia test equipment and applications.
Test Variables	A summary of the key test parameters that affect the test's performance and scale. These can be modified to construct other tests.
Results Analysis	Provides the background useful for test result analysis, explaining the metrics and providing examples of expected results.
Troubleshooting and Diagnostics	Provides guidance on how to troubleshoot common issues.
Conclusions	Summarizes the result of the test.

Typographic Conventions

In this document, the following conventions are used to indicate items that are selected or typed by you:

- **Bold** items are those that you select or click on. It is also used to indicate text found on the current GUI screen.
- *Italicized* items are those that you type.

Dear Reader

Ixia's Black Books include a number of IP and wireless test methodologies that will help you become familiar with new technologies and the key testing issues associated with them.

The Black Books can be considered primers on technology and testing. They include test methodologies that can be used to verify device and system functionality and performance. The methodologies are universally applicable to any test equipment. Step by step instructions using Ixia's test platform and applications are used to demonstrate the test methodology.

This tenth edition of the black books includes twenty two volumes covering some key technologies and test methodologies:

Volume 1 – Higher Speed Ethernet

Volume 2 – QoS Validation

Volume 3 – Advanced MPLS

Volume 4 – LTE Evolved Packet Core

Volume 5 – Application Delivery

Volume 6 – Voice over IP

Volume 7 – Converged Data Center

Volume 8 – Test Automation

Volume 9 – Converged Network Adapters

Volume 10 – Carrier Ethernet

Volume 11 – Ethernet Synchronization

Volume 12 – IPv6 Transition Technologies

Volume 13 – Video over IP

Volume 14 – Network Security

Volume 15 – MPLS-TP

Volume 16 – Ultra Low Latency (ULL) Testing

Volume 17 – Impairments

Volume 18 – LTE Access

Volume 19 – 802.11ac Wi-Fi Benchmarking

Volume 20 – SDN/OpenFlow

Volume 21 – Network Convergence Testing

Volume 22 – Testing Contact Centers

A soft copy of each of the chapters of the books and the associated test configurations are available on Ixia's Black Book website at http://www.ixiacom.com/blackbook. Registration is required to access this section of the Web site.

At Ixia, we know that the networking industry is constantly moving; we aim to be your technology partner through these ebbs and flows. We hope this Black Book series provides valuable insight into the evolution of our industry as it applies to test and measurement. Keep testing hard.

[illegible]

Errol Ginsberg, Acting CEO

Test Automation

Automation Methodologies

This document outlines methodologies and best practices for the automation of key components in an end-to-end test automation solution for IP systems. Following the methodologies in this guide will help to accelerate the testing cycle and increase test application engineer productivity and lab test tool equipment utilization.

Introduction

Test automation is a key ingredient to bringing switching and routing equipment to market quickly with high quality, minimal expense, and interoperability. Test automation makes it possible to perform far more tests than would be possible with manual testing. All aspects of the testing lifecycle, including feature test, system test, and regression test, can benefit from the enhanced consistency and increased speed derived from test automation.

The payoff for using test automation is substantial.

Network equipment manufacturers (NEMs) can use automation to:

- Automate regression tests for developers to ensure that nightly software builds haven't created more problems than they've solved.
- Combine tests from multiple vendors, platforms and applications during system integration.
- Automate regression tests for device and system quality assurance to insure that integrated systems continue to work together.
- Use regressions to ensure patches and updates haven't affected operation or performance.
- Track the result of regressions run over time to ensure that a project is making continuous progress toward zero bugs and optimal performance.

Service providers and enterprises can use automation to:

- Automate device qualification tests for both initial deployment and upgrades.
- Ensure that patches and updates haven't affected operation or performance.
- Ensure interoperability between current and prospective devices.
- Run daily tests to ensure that operation and performance haven't been affected by network, file system, or hacker activity.
- Reduce the effort required to create complex, real-world test cases.

Test Case: DUT CLI Automation by Using Test Composer

Overview

This tutorial will focus on the Test Composer advanced script authoring tool available in Test Conductor. Through the course of this exercise, you will use Test Composer to capture the interactions with a DUT over a Telnet session and record the CLI commands issued during that session. You will also learn how these recorded commands then become part of an automated test.

The tutorial follows a phased approach to create this script so that you can first learn the basics of the process, and then progress to add more script complexity until you achieve the final version. This approach to script development is the best way to approach all the scripting challenges because it makes it easier to develop bug-free tests in the shortest amount of time.

Objective

The ultimate goal of this tutorial is to show you to create a reusable procedure that can be shared across tests by many test engineers.

Setup

A Test Conductor Console is used to author a Test Composer script to automate a DUT's configuration. The Test Conductor Console communicates to the Test Conductor Server to store the test and to drive the execution of the test in unattended mode when it is included in a regression using the Test Conductor Regression Manager.

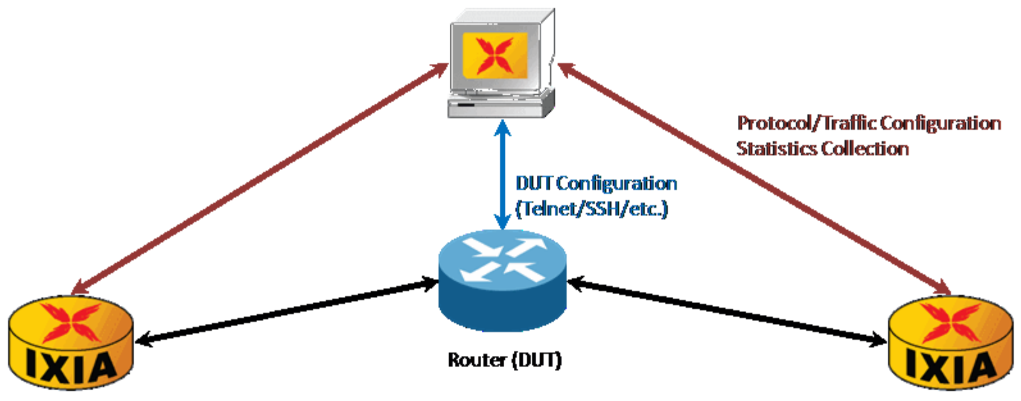


Figure 1. Composer Telnet CLI session logical diagram

Step-by-Step Instructions

1. Launch the application from the **Test Conductor** icon. Test Composer is a test authoring feature that is part of the Test Conductor suite.

   The GUI console window will appear. Test Composer is a new configuration tool available in the bottom left corner of the screen.

2. In this example, you will be starting with a blank test. Click the **Test Composer** icon. This is the entry point into the Test Composer script authoring module.

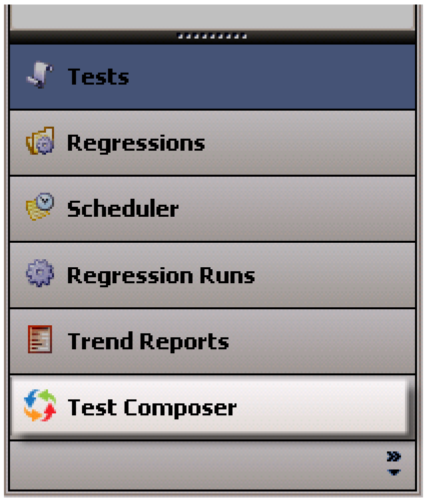


Figure 2. Launching Composer from the Test Conductor GUI console

3. Click the **New Procedure** icon to create the new procedure that will contain your script. Procedures are reusable blocks of scripting that can be used inside of tests or other procedures.

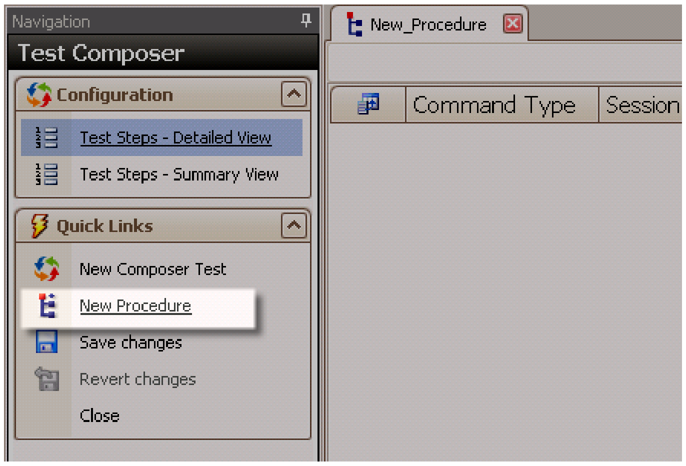
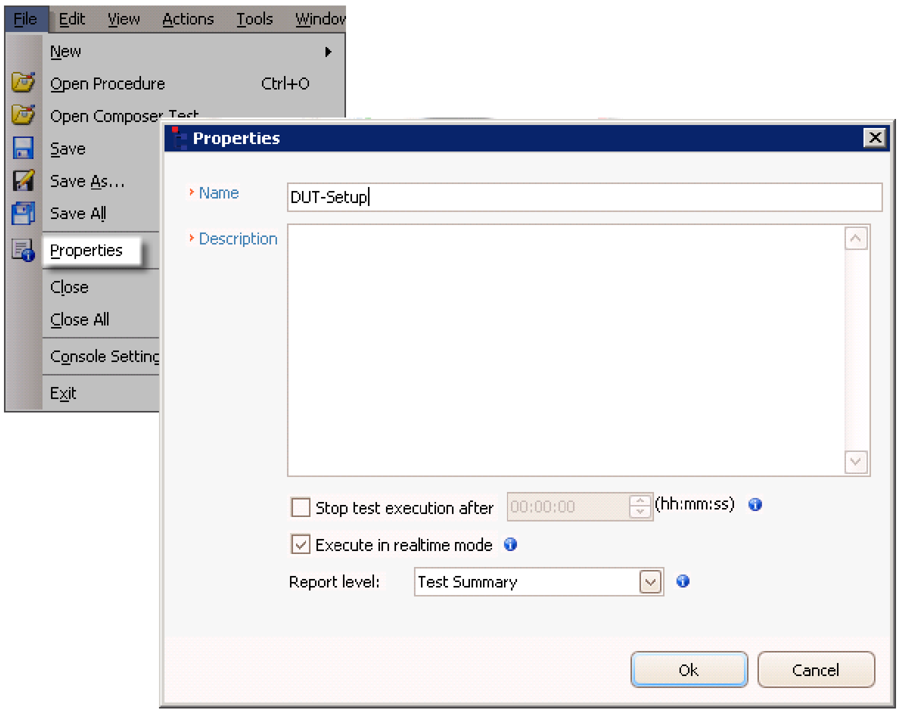


Figure 3. Creating a new Composer procedure

4. Set the properties of the procedure. You can do this now or later as part of saving your work. Change the **Name** field value to *DUT-Setup*. Leave the remaining options on this view with their default values.



Name: DUT-Setup
Description:
Stop test execution after 00:00:00 (hh:mm:ss)
Execute in realtime mode
Report level: Test Summary

Figure 4. Setting the Properties

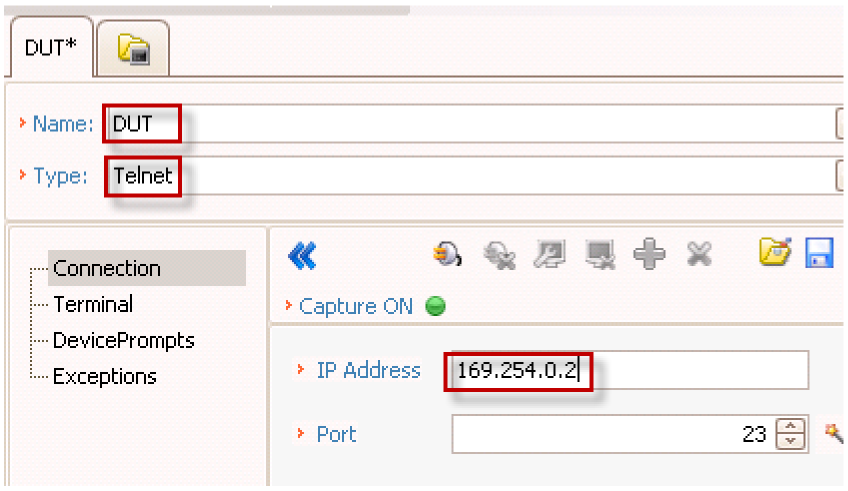
5. Click **Show Sessions Console** to make the Capture pane viewable.

6. (Optional) Change the **Name** field value to *DUT*. You may accept the default but we recommend that you specify a meaningful name so that the test is more understandable, and in cases where you need to have multiple sessions, it is easier to track what is happening when the names reflect the device being controlled.

7. Select **Telnet** from the **Type** list. This selects a connection type that matches the connection type for the device. Different devices use different session types, and each session type has a different set of input parameters.

8. Enter the IP Address *169.254.0.2* for your DUT in the **IP Address** field. This step is required for a Telnet session, but the IP Address will differ for each device connected.

9. Leave the default of *23* for the port. The only time you have to change the port number is when the device is using some non-standard port assignment. If your attempts to connect to the device time out, double-check the IP address and port to make sure they match with the device configuration.



Name: DUT
Type: Telnet
IP Address 169.254.0.2
Port 23

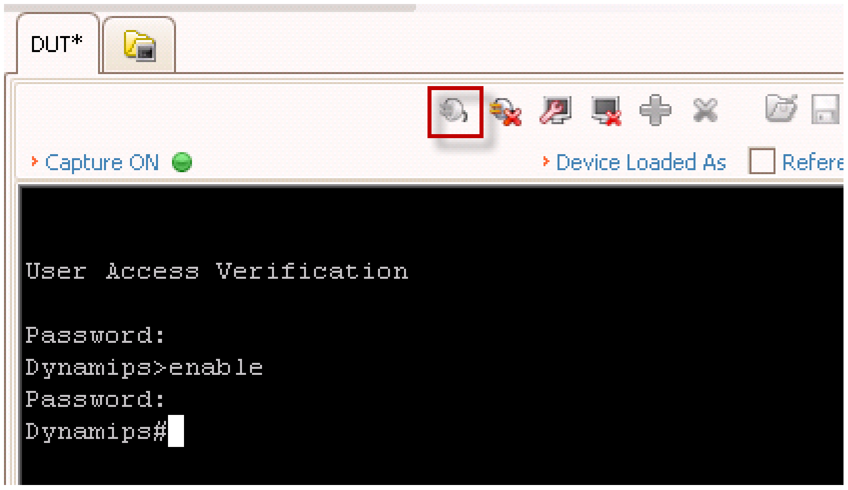
Figure 5. Creating a new Telnet session using the Session Manager

10. Click **Connect** to establish an active Telnet session to the DUT. This initializes the connection to the device. After you have connected, you should see a prompt from the device. However, sometimes the device needs to be 'awakened.' If you do not see a prompt, try clicking in the session window and pressing the ENTER key.

    Type directly into the Capture Window to issue commands directly to the device. These commands will be automatically captured as steps in the test editor.

11. Begin entering the commands you typically enter to configure the device. In the example, the first set of commands is required to log on to the device. This is an example – the details of the text will differ depending on what login information is required by the DUT.

    Password: *lab*
    >enable
    Password: *lab*
    # configure t

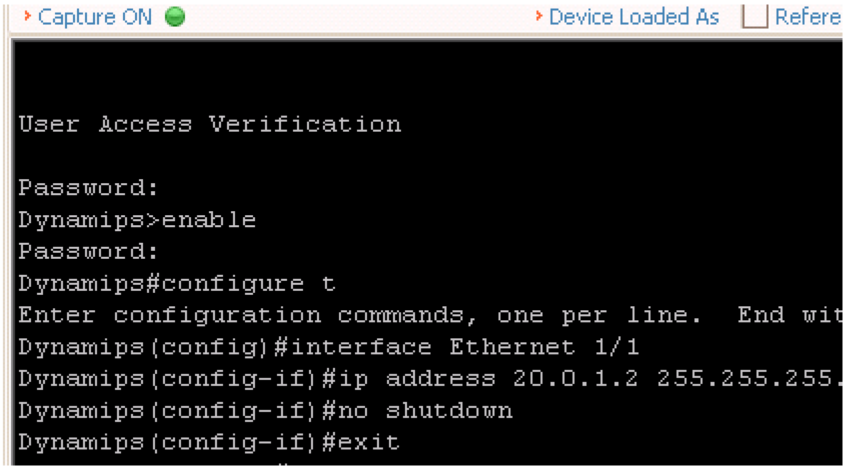


```
User Access Verification

Password:
Dynamips>enable
Password:
Dynamips#
```

Figure 6. Capturing a DUT login sequence using Session Manager

12. After you are past the first stage of logging on, you are ready to begin configuring the device. For example, type the following commands into the Capture Window to enable an interface on the device. The details of the text will differ depending on the type of device.

    (config)# *interface Ethernet 1/1*
        (config-if)# *ip address 20.0.1.2 255.255.255.0*
        (config-if)# *no shutdown*
        (config-if)# *exit*
        (config-)# *exit*



```
User Access Verification

Password:
Dynamips>enable
Password:
Dynamips#configure t
Enter configuration commands, one per line.  End wit
Dynamips(config)#interface Ethernet 1/1
Dynamips(config-if)#ip address 20.0.1.2 255.255.255.
Dynamips(config-if)#no shutdown
Dynamips(config-if)#exit
```

Figure 7. Capturing a DUT interface configuration sequence using Session Manager

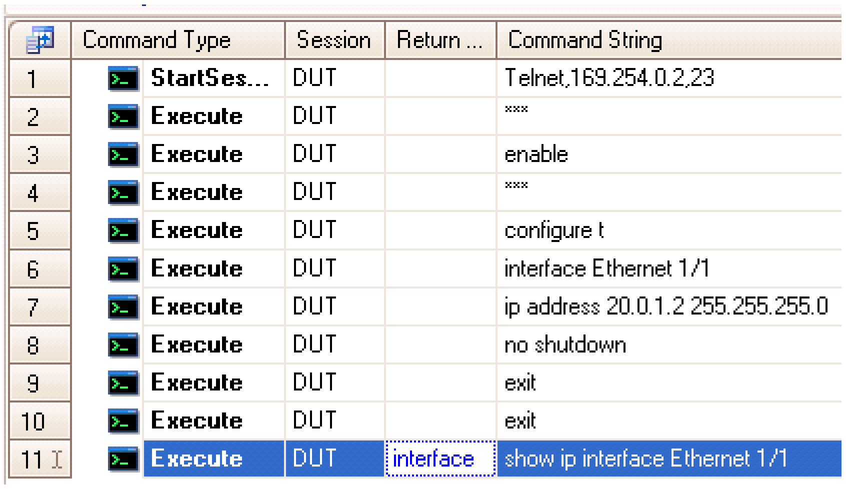
13. Commands can be used to configure the device or to get information back from the device. For example, you can use a status command like the one shown next to get information about the state of the configuration of the device. Type the commands into the Capture Window to generate a status response from the device. Note that the details of the text will differ depending on the type of device. (Figure 8)

    # show ip interface Ethernet 1/1
        --More-- <*Spacebar*>

14. Not only is the command being captured, but so is the response. To get the response back in the script where it can be analyzed, you have to give the response a name. Specifically, you need to provide the name of the variable that will store the response. Click the line with the show command you just entered, and then click the **Return Variable** field.

15. This part of the command is where the name of variable goes. In this example, the name of the variable to use is 'interface.' Enter the value *interface* for the Return Variable.

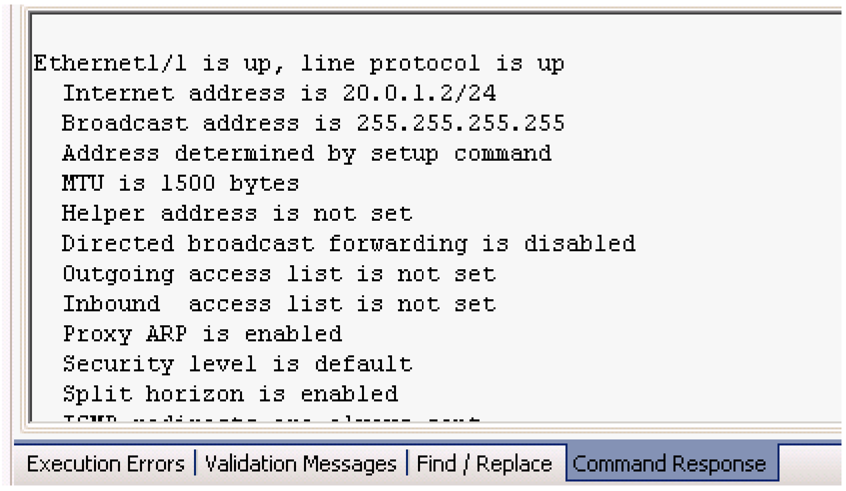
16. Press the ENTER key to accept.



	Command Type	Session	Return ...	Command String
1	StartSes...	DUT		Telnet,169.254.0.2,23
2	Execute	DUT		***
3	Execute	DUT		enable
4	Execute	DUT		***
5	Execute	DUT		configure t
6	Execute	DUT		interface Ethernet 1/1
7	Execute	DUT		ip address 20.0.1.2 255.255.255.0
8	Execute	DUT		no shutdown
9	Execute	DUT		exit
10	Execute	DUT		exit
11	Execute	DUT	interface	show ip interface Ethernet 1/1

Figure 8. Assigning a return variable to an Execute command

17. To see what response was captured for a command, click the **Command Response** tab to see the **Local Step Response** for the first execution of the *show ip interface Ethernet 1/1* command. The first time a command is executed, the response is captured. After that, the response is not captured. The captured response is used as a template to be compared to responses from the subsequent executions of the step. If the response is wrongly captured, or the command is modified in such a way as to change the fundamental structure of the response, you can delete the current capture, and then run the step again, which will capture a new version of the response.



```
Ethernet1/1 is up, line protocol is up
  Internet address is 20.0.1.2/24
  Broadcast address is 255.255.255.255
  Address determined by setup command
  MTU is 1500 bytes
  Helper address is not set
  Directed broadcast forwarding is disabled
  Outgoing access list is not set
  Inbound  access list is not set
  Proxy ARP is enabled
  Security level is default
  Split horizon is enabled
```

Figure 9. Viewing a free format text response in the Command Response tab

18. Sometimes, you do not want the whole response, but just some parts of it. Extracting the parts of interest is done using the Response Template Editor.Click **Edit** to launch the Response Template Editor. After the editor is open, you can identify which parts to extract and what to label the extracted parts.

19. Different kinds of responses have different structures to them. Some responses have a static structure, while others have a dynamic structure. Using the right method of extraction is key to the repeatability of the test. In this example, you will use the regular expression form of extraction. Click the **Regex target** icon to create a Free Format Target. This is used to match a text variable against a unique string pattern.

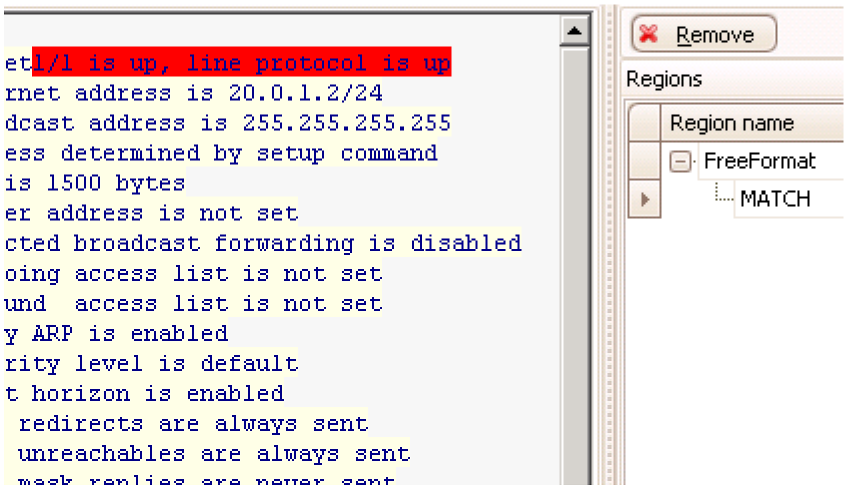
20. (Optional) A meaningful label for the target makes it easier to read and understand the test. To give the target a proper label, change the **Target Name** property to something that reflects the content of the target. In this example, you can use the word *MATCH*. The default name may also be used.

21. Different regular expressions will find different parts of the response. A variety of sample regular expressions, called expression templates are provided to make it easier to extract the parts of the response you want. Change the Regular expression template property to *C1 [0-9a-zA-Z]**. The C1 property is used to identify the key to finding the text of interest and the string of text following it describes the pattern of characters that represent the part of the response that you want to extract. In this example, it indicates any length and mix of characters (both upper and lower case) and numbers.

22. The expression templates are sometimes close to what you need, but not quite. You can change the expression to fit exactly what you need. In this example, replace only the plus sign (+) with an asterisk (*) in the group *(?<md>[\s]+)* in the pattern defined for the regular expression. This step is required in this example, but is optional in general.

23. Change the value of the **C1** property to *Ethernet because in this response, the word Ethernet represents the start of the text to extract*. This is done when a portion of the text response is a fixed string of text (for example, *Ethernet*) that always remains the same in the pattern.

24. Click **OK** to save the response mapping. The result will also show in the Command Response window. This saves the mapping.



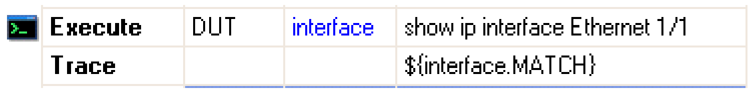
```
et1/1 is up, line protocol is up
rnet address is 20.0.1.2/24
dcast address is 255.255.255.255
ess determined by setup command
is 1500 bytes
er address is not set
cted broadcast forwarding is disabled
oing access list is not set
und  access list is not set
y ARP is enabled
rity level is default
t horizon is enabled
 redirects are always sent
 unreachables are always sent
```

Figure 10. Matching free format text strings in the Response Map Editor

25. Not all the commands that are in your procedure have to be created by capturing them in the session window. Some commands are entered directly into the script. In this example, you will be adding a command to display to the screen the value that is extracted from the response. Highlight the line that you were just working on and click **Insert Step Below** to create a new line. This manually inserts a step and the type of command inserted is the same type as the one last inserted. If no command has been manually inserted yet, by default, a Comment command is inserted.

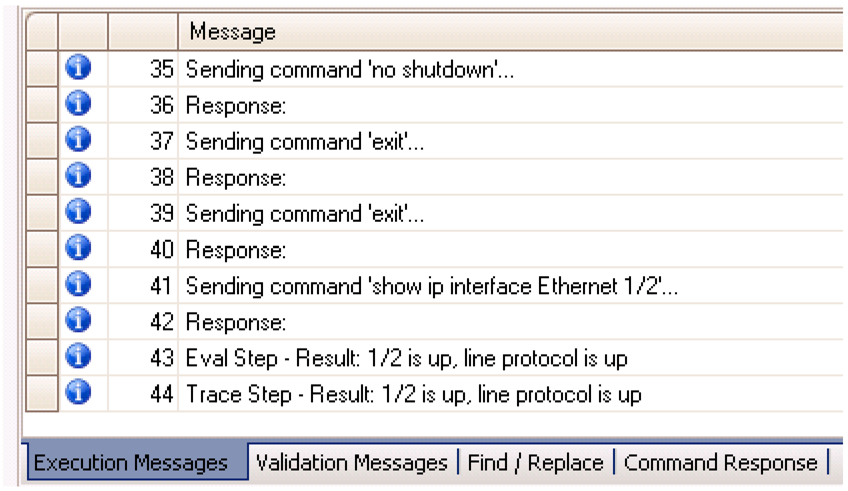
26. Click the **Command Type** list to change the **Command Type** to **Trace**.

27. You can modify the command string directly by clicking in the **Command String** field and entering text directly. To make it easier to edit the command string, you can use the variable text completion feature to change the value of the command string to *${interface.MATCH}*. This specifies an output argument for the **Trace** command.



Execute	DUT	interface	show ip interface Ethernet 1/1
Trace			${interface.MATCH}

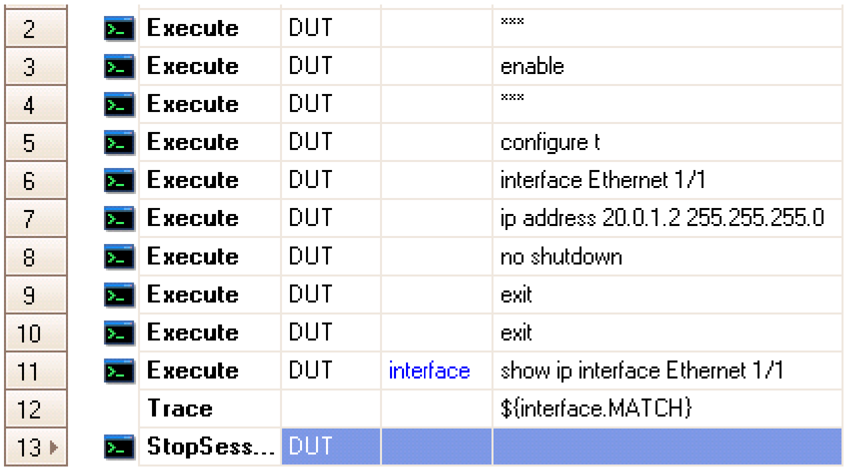
Figure 11. Using Variables in commands



```
   Message
35 Sending command 'no shutdown'...
36 Response:
37 Sending command 'exit'...
38 Response:
39 Sending command 'exit'...
40 Response:
41 Sending command 'show ip interface Ethernet 1/2'...
42 Response:
43 Eval Step - Result: 1/2 is up, line protocol is up
44 Trace Step - Result: 1/2 is up, line protocol is up
```

Figure 12. Logging information messages in the Execution Messages tab

28. After you are done configuring the DUT, you need to close the session. Click **Disconnect** to close the Telnet session. This will automatically create a line with a **Command Type** of **StopSession**. This step will disconnect the session from the DUT whenever it is executed.



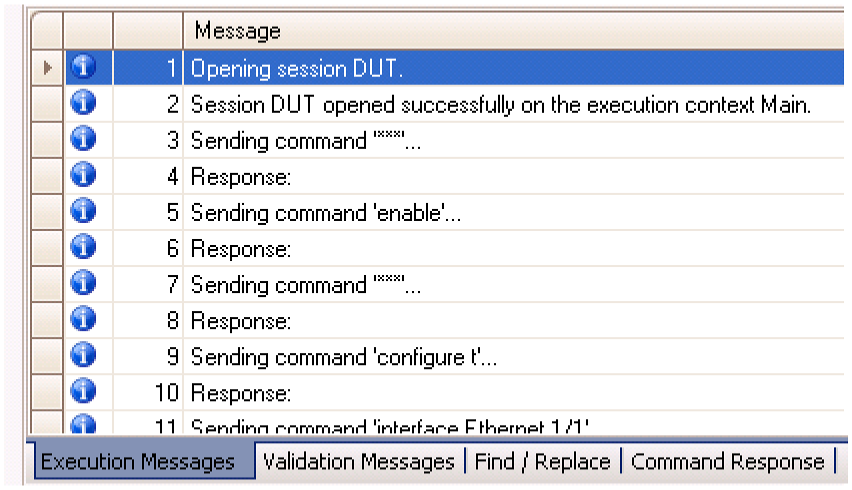
2	Execute	DUT		***
3	Execute	DUT		enable
4	Execute	DUT		***
5	Execute	DUT		configure t
6	Execute	DUT		interface Ethernet 1/1
7	Execute	DUT		ip address 20.0.1.2 255.255.255.0
8	Execute	DUT		no shutdown
9	Execute	DUT		exit
10	Execute	DUT		exit
11	Execute	DUT	interface	show ip interface Ethernet 1/1
12	Trace			${interface.MATCH}
13	StopSess...	DUT		

Figure 13. Inserting an Automatic StopSession step on session disconnect

29. To replay your captured steps, you can highlight one or more of the steps and press play. Highlight line #1 through line #13.

30. Click **Play** to replay the captured CLI commands to the DUT. This replays captured commands back to a session.

31. As the commands execute, status messages will be appear in the **Execution Messages** window. This will verify that replayed commands are executing correctly.



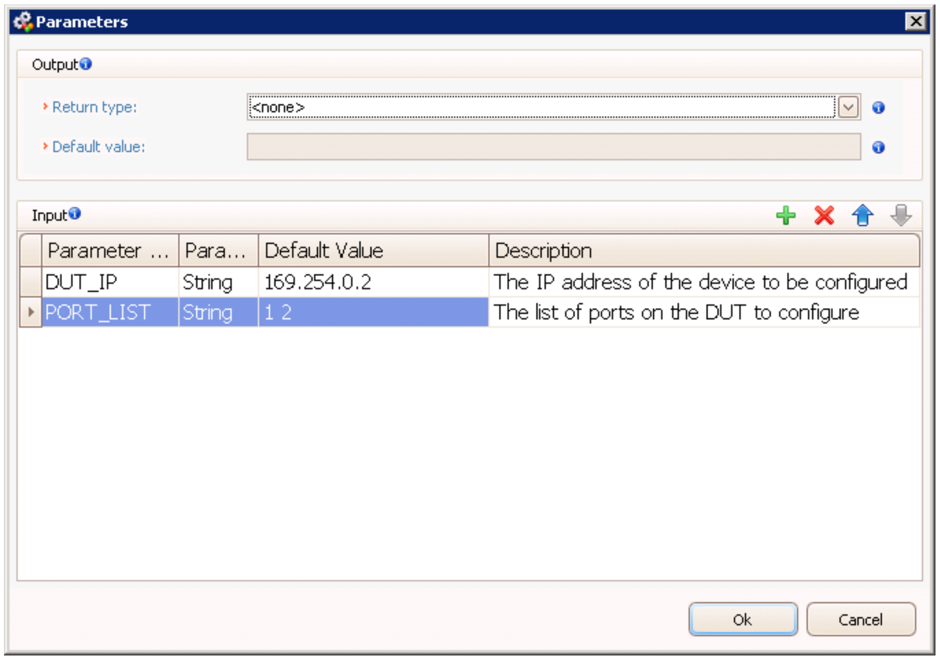
	Message
1	Opening session DUT.
2	Session DUT opened successfully on the execution context Main.
3	Sending command '***'...
4	Response:
5	Sending command 'enable'...
6	Response:
7	Sending command '***'...
8	Response:
9	Sending command 'configure t'...
10	Response:
11	Sending command 'interface Ethernet 1/1'

Figure 14. Logging execution information messages in the Execution Errors tab

The sequence of steps that have been created so far are a good start to create a useful procedure intended to configure a DUT. However, it is not very portable. The DUT's address, for example, is hard-coded in the procedure. To make the procedure more reusable across tests and DUTs, it would be better if the address could be configurable when the procedure is called by some test. The next few steps show how to parameterize the procedure for the address of the DUT and the list of ports to configure.

1. To add a new parameter to the procedure, click **Parameters** to open the **Parameters** window.
2. Click **Add Parameter** to create a new row for first input parameter. This will be used to add an input variable to the list.
3. Change the **Parameter Name** to *DUT_IP*. Set the **Default Value** to *169.254.0.2*. This is a common global variable needed for this example. Default values are useful when you are debugging the procedure because they allow you to run the procedure without having to call the procedure from an external procedure and pass in the values for the parameters being used.
4. (Optional) A good practice is to include a description for the parameter. This helps when someone else wants to use your procedure. When they make a call to your procedure and open the parameter list, they will see your descriptions and know what values they need to provide without having to open the actual procedure to understand how the parameter is used.
5. Click **Add Parameter** to create a new row for the second input parameter.
6. Change the **Parameter Name** to *PORT_LIST*.
7. Set the **Default Value** to the space-separated port list *1 2*. This is good practice to initialize all global variables to known good values.

8. Click **OK** to accept your two input parameters and save the new global variables.



Parameters

Output

Return type: <none>

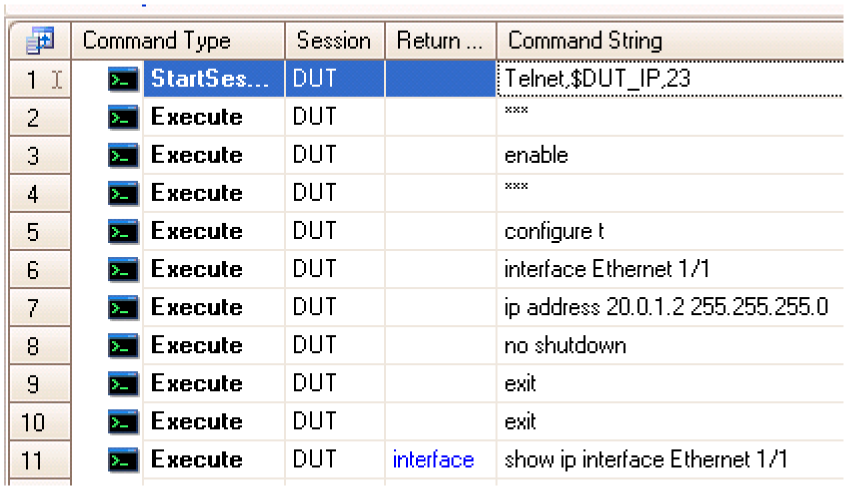
Default value:

Input

Parameter ...	Para...	Default Value	Description
DUT_IP	String	169.254.0.2	The IP address of the device to be configured
PORT_LIST	String	1 2	The list of ports on the DUT to configure

Figure 15. Adding input parameters for variable fields in Execute commands

9. Click line #1 and click the **Command String** field to change the **IP Address** from a hard-coded value to the variable *$DUT_IP*. This changes a fixed value to a dynamic value.

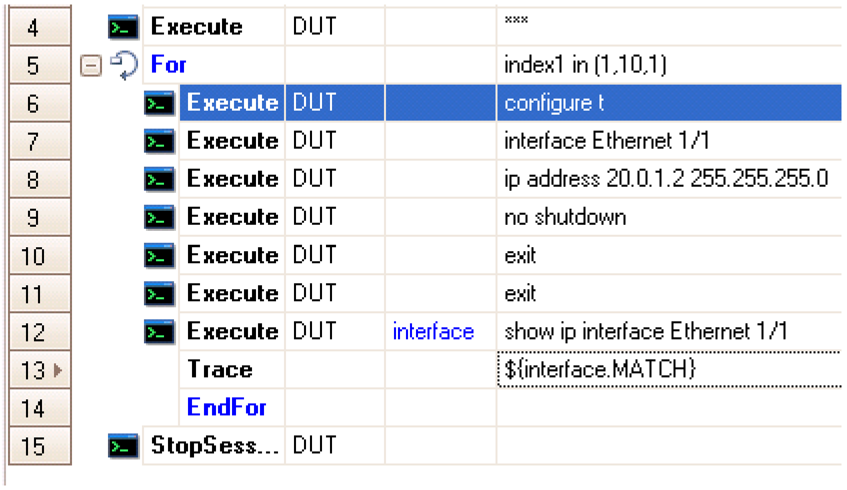


	Command Type	Session	Return ...	Command String
1	StartSes...	DUT		Telnet,$DUT_IP,23
2	Execute	DUT		***
3	Execute	DUT		enable
4	Execute	DUT		***
5	Execute	DUT		configure t
6	Execute	DUT		interface Ethernet 1/1
7	Execute	DUT		ip address 20.0.1.2 255.255.255.0
8	Execute	DUT		no shutdown
9	Execute	DUT		exit
10	Execute	DUT		exit
11	Execute	DUT	interface	show ip interface Ethernet 1/1

Figure 16. Parameterization of device IP address in StartSession

10. Because your test is currently designed to configure only one interface, it is not very flexible. Some tests may need to configure multiple interfaces. By taking the commands used to configure one interface and insert them into a loop, the test can configure any number of interfaces. And, because the test has an input parameter that takes a list of interfaces, the test using your procedure can decide which interfaces need to be configured by simply passing them to the procedure. Highlight line #5 through line #12so that they can be enclosed inside of a loop.

11. Click the **Place Inside For** icon. Use the list to choose between different loop types for insertion. To make the test easier to read, you will see with steps indent inside a new **For** loop beginning at line #5 and ending at line #14. This encloses your steps inside the loop that will configure each interface in the list. The same set of lines will be executed many times with slightly different values.



4	Execute	DUT		""
5	For			index1 in (1,10,1)
6	Execute	DUT		configure t
7	Execute	DUT		interface Ethernet 1/1
8	Execute	DUT		ip address 20.0.1.2 255.255.255.0
9	Execute	DUT		no shutdown
10	Execute	DUT		exit
11	Execute	DUT		exit
12	Execute	DUT	interface	show ip interface Ethernet 1/1
13	Trace			${interface.MATCH}
14	EndFor			
15	StopSess...	DUT		

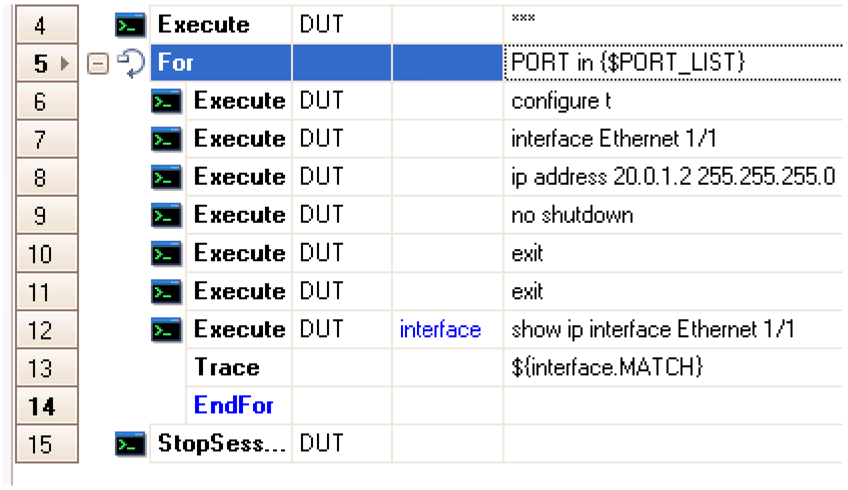
Figure 17. Encapsulating repeating Execute commands in control flow logic

12. Click the **Command String** list to open the **For** loop expression builder. Enter *PORT* for the value of the **Assign** to variable field. This changes the name of the variable that will be used to parameterize your commands .

13. Select **SET** from the **Loop** type list. This tells the command that it will be using a list of values instead of a range of values. Sometimes, this is called a ForEach style of loop.

14. Click **Select Variable** and check **Use variable** to specify *PORT_LIST* as the variable for the **Values** field. This step is required whenever a dynamic list is needed as opposed to a fixed set of values.

15. Click **Check** to accept your changes. This step is required to save changes made to the loop expression.

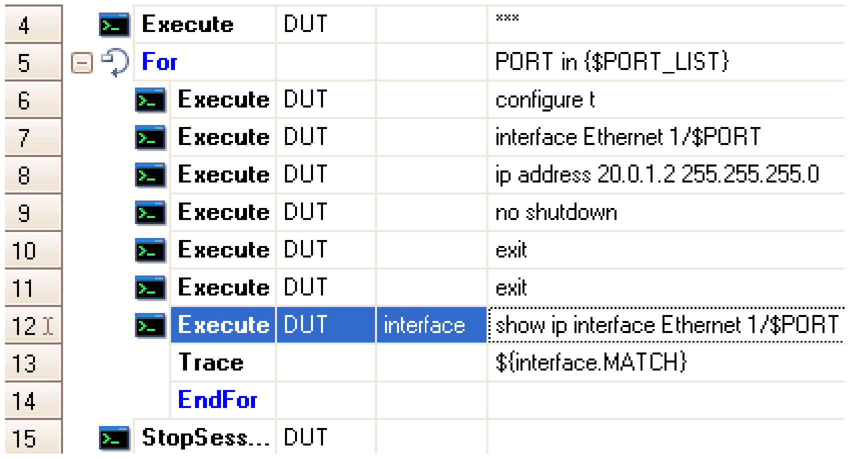


4	Execute	DUT		***
5	For			PORT in {$PORT_LIST}
6	Execute	DUT		configure t
7	Execute	DUT		interface Ethernet 1/1
8	Execute	DUT		ip address 20.0.1.2 255.255.255.0
9	Execute	DUT		no shutdown
10	Execute	DUT		exit
11	Execute	DUT		exit
12	Execute	DUT	interface	show ip interface Ethernet 1/1
13	Trace			${interface.MATCH}
14	EndFor			
15	StopSess...	DUT		

Figure 18. Iteration over a fixed set of values contained in a variable

16. Change the hard-coded port value in line #7 to use the parameter specified in the FOR command so that the **Command String** value is *interface Ethernet 1/$PORT*. This step is an example. Changing hard-coded values to variables is the same process regardless of the command.

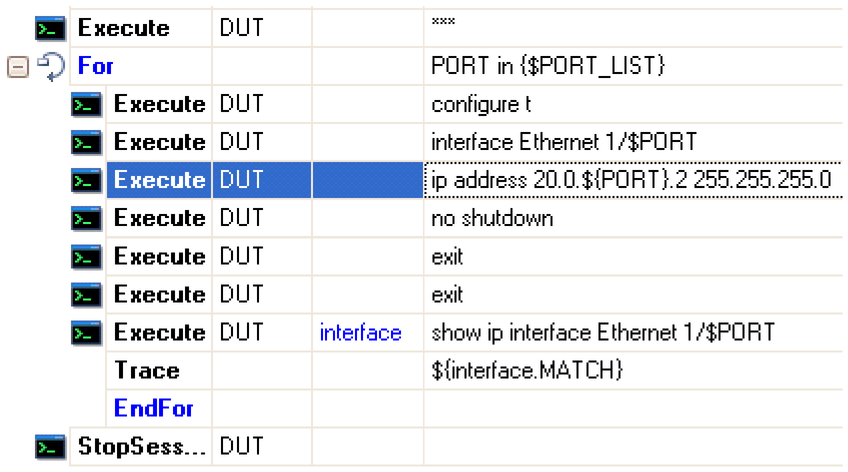
17. Change the hard-coded port value in line #12 to a parameter just like the previous step so that the command looks like *show ip interface Ethernet 1/$PORT*.



4	Execute	DUT		***
5	For			PORT in {$PORT_LIST}
6	Execute	DUT		configure t
7	Execute	DUT		interface Ethernet 1/$PORT
8	Execute	DUT		ip address 20.0.1.2 255.255.255.0
9	Execute	DUT		no shutdown
10	Execute	DUT		exit
11	Execute	DUT		exit
12	Execute	DUT	interface	show ip interface Ethernet 1/$PORT
13	Trace			${interface.MATCH}
14	EndFor			
15	StopSess...	DUT		

Figure 19. Parameterization of variable portions of an Execute command

18. Change the third octet value in the **IP Address** in line #8 so it looks like *ip address 20.0.${PORT}.2 255.255.255.0*.
Note that braces are needed in this case so that the interpreter can distinguish the parameter as a Tcl variable when dots are part of the command string value.

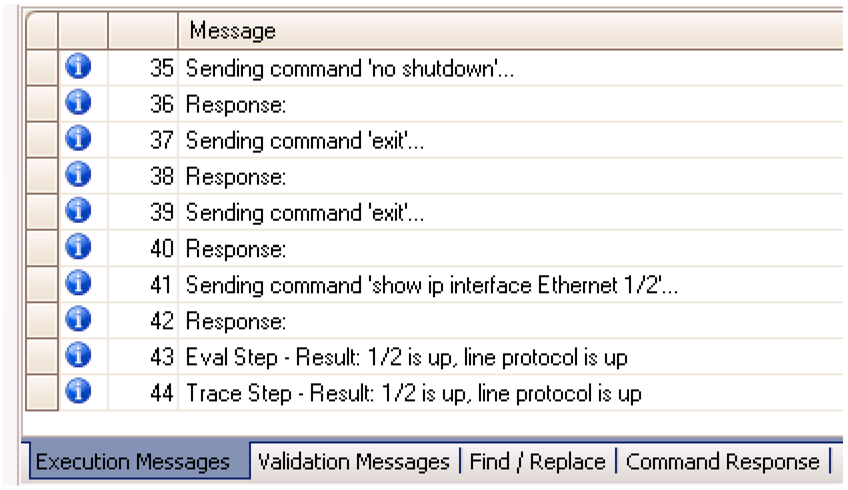


Execute	DUT		***
For			PORT in {$PORT_LIST}
Execute	DUT		configure t
Execute	DUT		interface Ethernet 1/$PORT
Execute	DUT		ip address 20.0.${PORT}.2 255.255.255.0
Execute	DUT		no shutdown
Execute	DUT		exit
Execute	DUT		exit
Execute	DUT	interface	show ip interface Ethernet 1/$PORT
Trace			${interface.MATCH}
EndFor			
StopSess...	DUT		

Figure 20. Parameterization of variable portions of an Execute command

19. Highlight line #1 through line #14. Click **Play** to send the CLI commands to the DUT and replay these captured steps back to the session.

20. As the commands execute, status messages will appear in the **Execution Messages** window. Note that as the commands run, the steps are repeated for each item in the PORT_LIST variable.



		Message
●	35	Sending command 'no shutdown'...
●	36	Response:
●	37	Sending command 'exit'...
●	38	Response:
●	39	Sending command 'exit'...
●	40	Response:
●	41	Sending command 'show ip interface Ethernet 1/2'...
●	42	Response:
●	43	Eval Step - Result: 1/2 is up, line protocol is up
●	44	Trace Step - Result: 1/2 is up, line protocol is up

Figure 21. Logging messages generated upon iterative execution of control flow logic

21. (Optional) In some situations, you may want to send commands to the DUT without having the commands captured in the test. It is not a problem if unwanted commands are captured because you can delete them at any time. But if you want to avoid having to delete unwanted commands, click **Capture Off** to toggle capture mode to off as shown in Figure 22. This allows you to experiment with commands without recording the session.

22. Click **Disconnect** to close the Telnet session; this time no step will be created. This disconnects the session. Since this is the second time this session has been disconnected, doing so with **Capture Off** ensures that no additional **StopSession** command is added to the script.

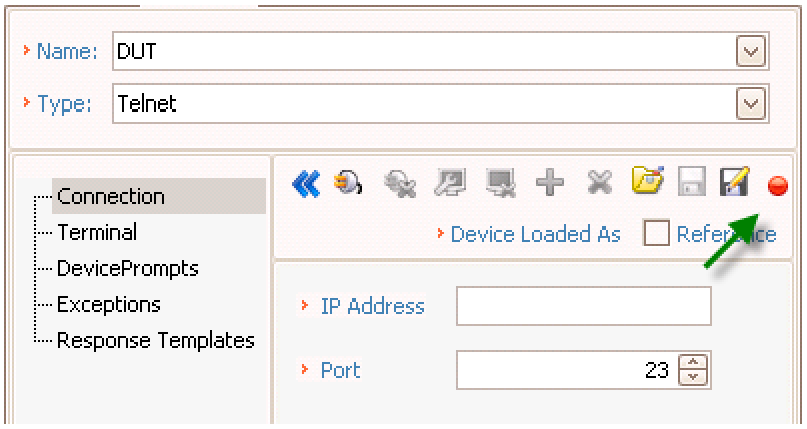
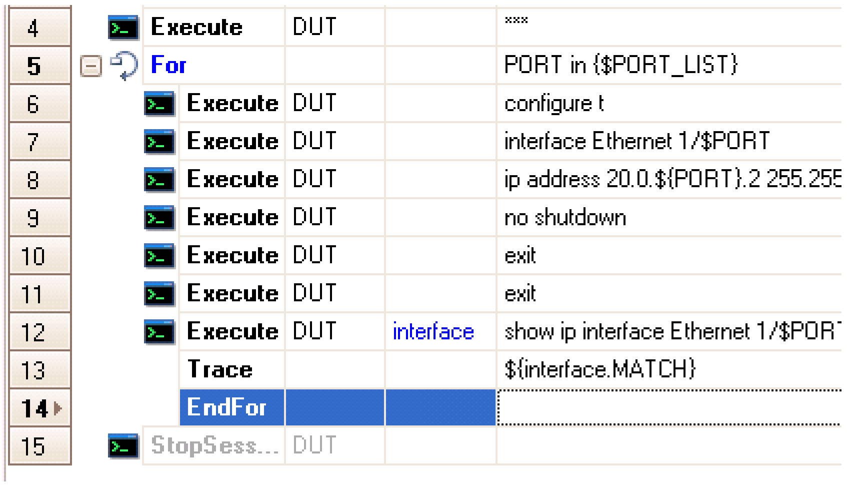


Figure 22. Disabling capture of DUT command line interface text

Sometimes, there are commands you do not want to execute, but you also do not want to remove them because they will be used again later. You can exclude a step from running by marking it as excluded.

1. Click line #15. Right-click to open the context menu and select **Exclude** to disable the execution of **StopSession** command. The **Command Type** fields for excluded commands are grayed out.

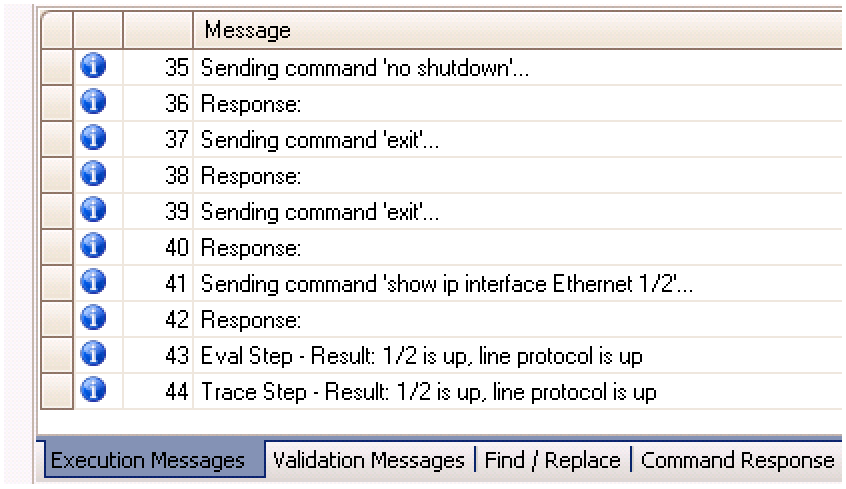


4	Execute	DUT		"""
5	For			PORT in {$PORT_LIST}
6	Execute	DUT		configure t
7	Execute	DUT		interface Ethernet 1/$PORT
8	Execute	DUT		ip address 20.0.${PORT}.2 255.255
9	Execute	DUT		no shutdown
10	Execute	DUT		exit
11	Execute	DUT		exit
12	Execute	DUT	interface	show ip interface Ethernet 1/$POR
13	Trace			${interface.MATCH}
14	EndFor			
15	StopSess...	DUT		

Figure 23. Excluding a StopSession command from the list of active commands

2. Highlight line #1 through line #15. Click **Play** to send the CLI commands to the DUT.

3. As the commands execute, status messages will appear in the **Execution Messages** window. Only commands marked as **Include** will be executed, therefore **your session will NOT be** automatically disconnected.

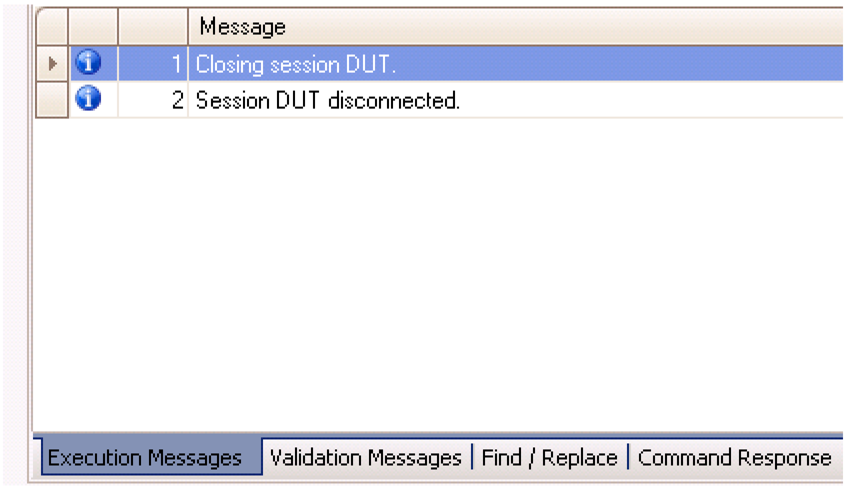


```
   Message
35 Sending command 'no shutdown'...
36 Response:
37 Sending command 'exit'...
38 Response:
39 Sending command 'exit'...
40 Response:
41 Sending command 'show ip interface Ethernet 1/2'...
42 Response:
43 Eval Step - Result: 1/2 is up, line protocol is up
44 Trace Step - Result: 1/2 is up, line protocol is up
```

Figure 24. Logging execution information messages in the Execution Errors tab

4. Click line #15. Right-click to open the context menu and select **Include** to re-enable the execution of the **StopSession** command.

5. Click **Play** to send the **StopSession** command that will terminate the session. This allows script execution to be broken up into pieces while in the editor.



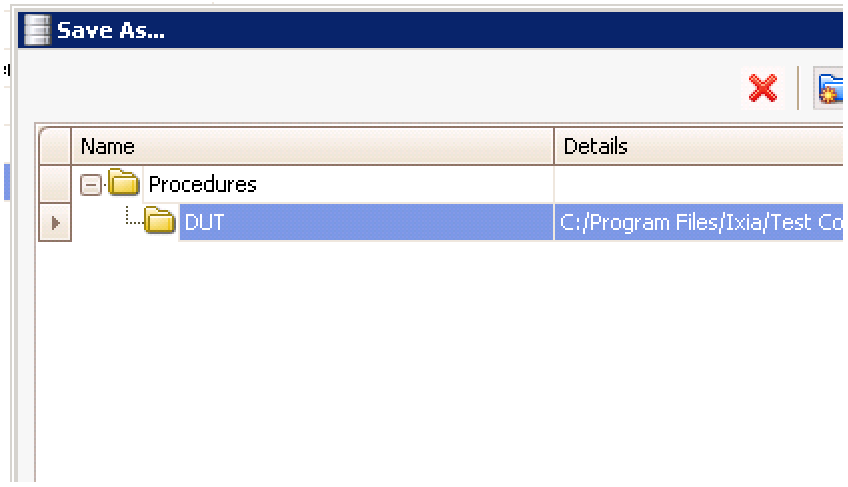
```
  Message
1 Closing session DUT.
2 Session DUT disconnected.
```

Figure 25. Logging session disconnect messages in the Execution Errors tab

6. Click **Save** to save the final version of your DUT CLI Telnet script.

In some cases, you may want to use an existing test as the starting point for a new procedure. You may also want to start organizing your procedures in some logical set of folders.

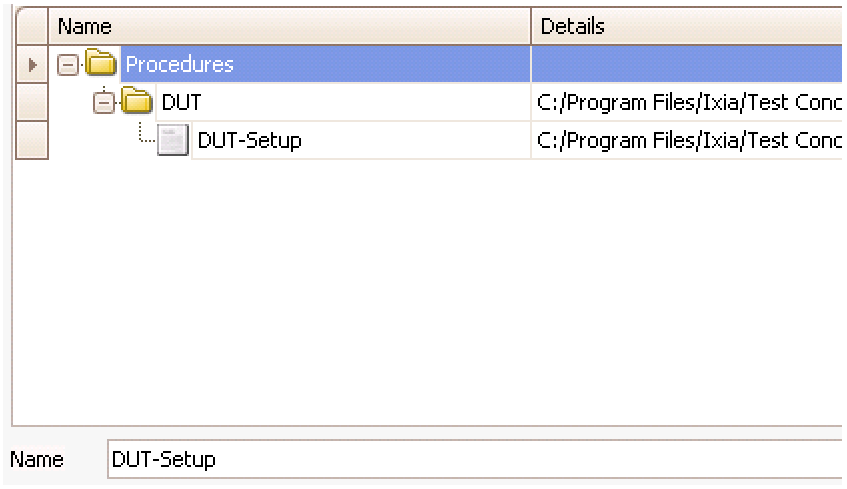
1. The easiest way to do this is to open the procedure that you want to copy and choose Save As. In the **Save As** window, click **New Folder** and create a new folder named *DUT* under the **Procedures** folder.



Name	Details
Procedures	
DUT	C:/Program Files/Ixia/Test Co

Figure 26. Creating a container folder for storing DUT procedures as resources

2. If you have not set the name of your procedure using the Properties, you can specify a procedure **Name** of *DUT-Setup* and then place this new version under the **DUT** folder under **Procedures** that you created in the previous step.



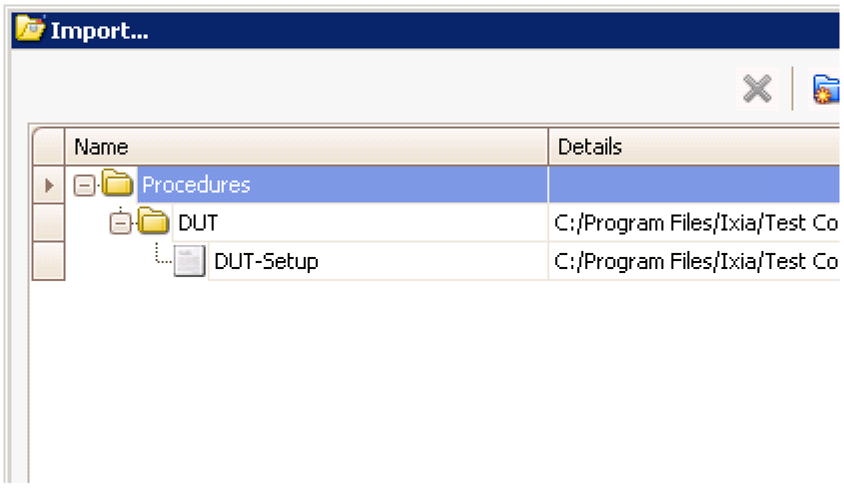
Name	Details
Procedures	
DUT	C:/Program Files/Ixia/Test Conc
DUT-Setup	C:/Program Files/Ixia/Test Conc

Name: DUT-Setup

Figure 27. Saving a DUT procedure as a reusable resource

Sometimes, you may prefer to make a copy of a procedure in your test rather than just calling the procedure remotely. In this case, the easiest way to include a procedure in a new test is to import the procedure into your test.

1. Start with a new test. Click the **New Composer Test** icon to create a new Composer test.
2. (Optional) Change the **Name** field value to *DUT-Lab*. It is recommended that a meaningful name be used. Leave the remaining options on this view at their default values. The remaining options are typically left at their defaults.
3. Click **Import Procedure** to import a procedure that was saved as a shared resource.
4. Select the **DUT-Setup** procedure you created previously and click **OK** to continue. This step completes the import process.

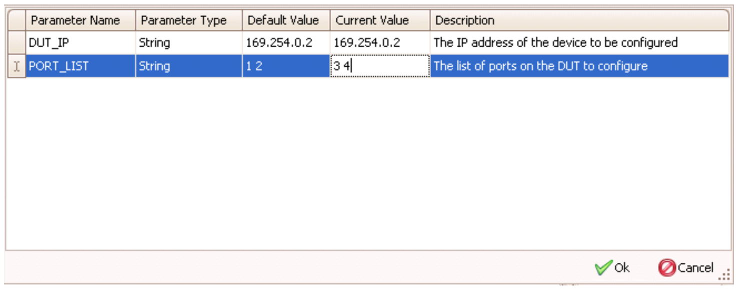


Name	Details
Procedures	
DUT	C:/Program Files/Ixia/Test Co
DUT-Setup	C:/Program Files/Ixia/Test Co

Figure 28. Browsing for DUT procedure resources from Import Procedure menu

5. Click **Append Last Step** to place a step at the end of the main body and before any procedures. This inserts a line at the end of a script's main body.
6. Click the **Command Type** list to change the **Command Type** to **RunProcedure**. Select the DUT-Setup as the **Local Procedure.**

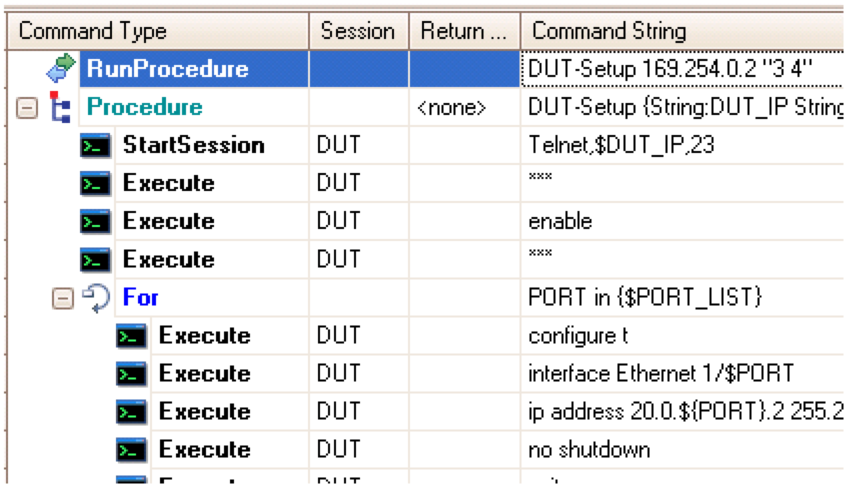
7. Click the **Command String** list to open the argument builder. This allows you to specify the arguments to a procedure. Specify *169.254.0.2* as the first argument, and then the space-separated value *3 4* as the second argument in the **Input Arguments** list. This example shows that the value of one or more variables may be changed when the procedure is called. The set of device ports to be configured are arguments to the procedure, and thus may be set to a different set of ports.



Parameter Name	Parameter Type	Default Value	Current Value	Description
DUT_IP	String	169.254.0.2	169.254.0.2	The IP address of the device to be configured
PORT_LIST	String	1 2	3 4	The list of ports on the DUT to configure

Figure 29. Inputting the values for the parameters of a procedure

8. Click **OK** to accept your changes. This step is required to save all work.



Command Type	Session	Return ...	Command String
RunProcedure			DUT-Setup 169.254.0.2 "3 4"
Procedure		<none>	DUT-Setup {String:DUT_IP String
StartSession	DUT		Telnet,$DUT_IP,23
Execute	DUT		***
Execute	DUT		enable
Execute	DUT		***
For			PORT in {$PORT_LIST}
Execute	DUT		configure t
Execute	DUT		interface Ethernet 1/$PORT
Execute	DUT		ip address 20.0.${PORT}.2 255.2
Execute	DUT		no shutdown

Figure 30. RunProcedure call to execute a DUT setup command with arguments

9. Highlight line #1.

10. Click **Play** to send the CLI commands to the DUT.

11. As the commands execute, status messages will appear in the **Execution Messages** window.

Test Case: Ixia Traffic Generator Automation by Using Test Composer

Overview

This tutorial will focus on the Test Composer advanced script-authoring tool available in Test Conductor. Through the course of this exercise, you will use Test Composer to author an IxExplorer script. The tutorial follows a phased approach to creating this script so that you can follow the process from the beginning, to adding more script complexity, to the final version.

Objective

The ultimate goal of this tutorial will be to apply the traffic configuration stored in a ScriptGen Tcl file to a set of ports, which will automate the transmission of traffic as well as statistics collection. The statistics can be used as final results, as well as for decision points during the test, directing execution flow.

In the second portion of this tutorial, you will continue with the IxExplorer example, but making the test more dynamic. You will add **if-statements** for decision making and loops for control flow. Finally, you will create additional sessions to collect port logs and organize steps into procedures.

Setup

A Test Conductor Console is used to author a Test Composer script that will automate configuration commands of an Ixia traffic generator. The Test Conductor Console communicates to the Test Conductor Server to store the test and to drive the execution of the test in unattended mode when it is included in a regression using the Test Conductor Regression Manager.

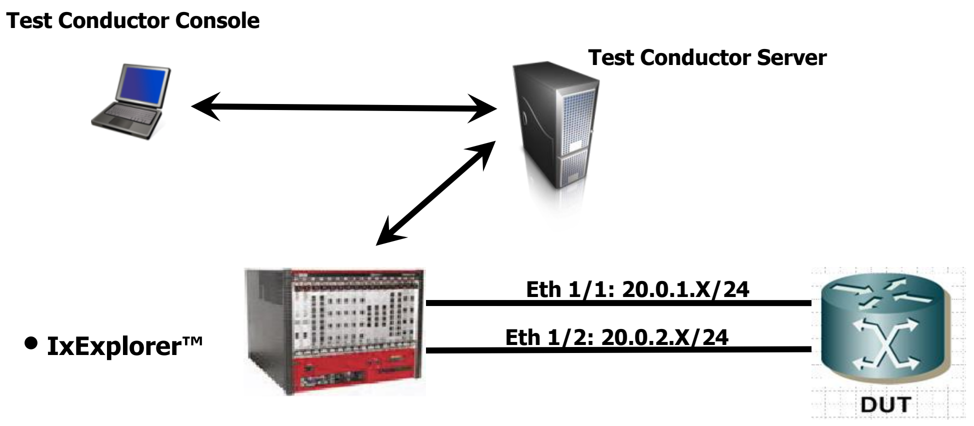


Figure 31. Composer Ixia Traffic Generator Session Logical Diagram

Step-by-Step Instructions

1. Launch the application from the **Test Conductor** Icon. Test Composer is a module within the Test Conductor suite.

   The GUI Console window will appear. This is the client window where all interaction with Test Conductor occurs.

2. In this example, you will be starting with a blank test. Click the **Test Composer** icon. This is the entry point into the Test Composer script authoring module.

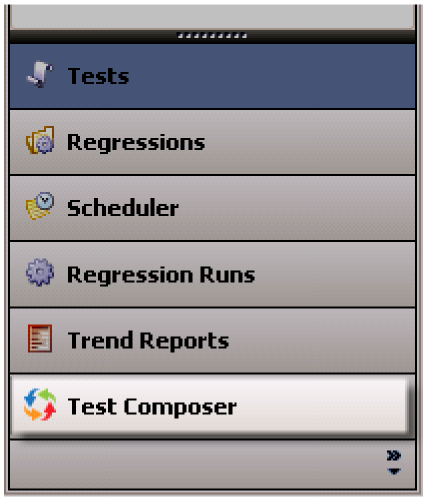


Figure 32. Launching Composer from within the Test Conductor GUI Console

3. Click the **New Composer Test** icon to create a test which will contain multiple procedures as opposed to a single procedure.

4. (Optional) Change the **Name** field value to *IxExplorer-Lab*. It is recommended that a meaningful name be chosen.

   The remaining settings on this view are typically left at their default values.

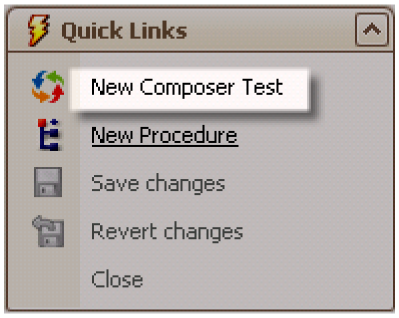


Figure 33. Creating a new Composer test

5. Click **Show Sessions Console** to display the **Capture** pane.

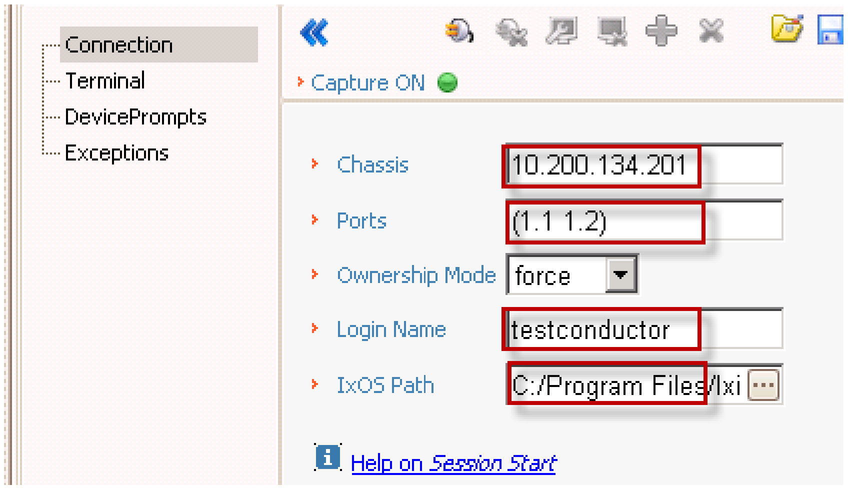
6. Change the **Name** field value to *XM2*. Select **IxExplorer** from the **Type** list. This identifies the session type for the type of device to be connected.

7. Enter the **IP Address** or **Hostname** for your chassis in the **Chassis** field. This is required for an IxExplorer chassis connection.

8. Enter the IDs of the two ports to which you will apply stream configuration into the **Ports** field using (C.P1 C.P2) notation where C = card #, P1 = port1 #, P2 = port2 #. For example: *1.1 1.2*. This step is required as ports must be owned in order to completely establish a session.

9. (Optional) Change the **Login Name** value to *testconductor*. It is considered good practice to take ownership of ports with a unique name.

10. (Optional) Browse for the top level **IxOS Path** for the version of IxOS you are using. If no IxOS version is selected, the connection will attempt to determine the default path. Keep in mind that you may be using multi-version IxOS. Example: *C:/Program Files/Ixia/IxOS/5.20.401.68-EB*. This choice is an example only and will differ depending on which IxOS is installed on the client machine.

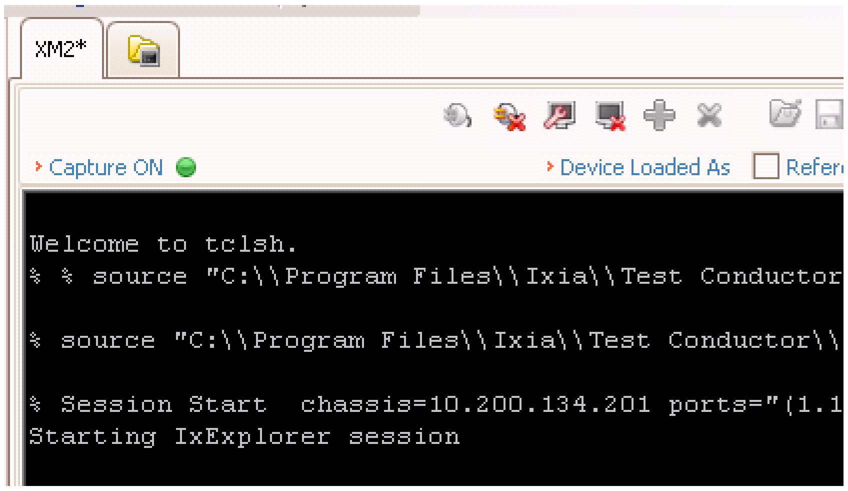


Chassis: 10.200.134.201
Ports: (1.1 1.2)
Ownership Mode: force
Login Name: testconductor
IxOS Path: C:/Program Files/Ixi

Figure 34. Connection parameters for an Ixia Traffic Generator session

11. Click the **Connect** icon to establish a new, active session to the Ixia Chassis Tcl Server. The IxExplorer session will take ownership of the ports you provided and generate a message in the **Capture** window indicating success or failure.

12. In the **Test Steps** window, line #1 has been automatically created with a **Command Type** of **StartSession**. Click **Insert Step Below** to create line #2.

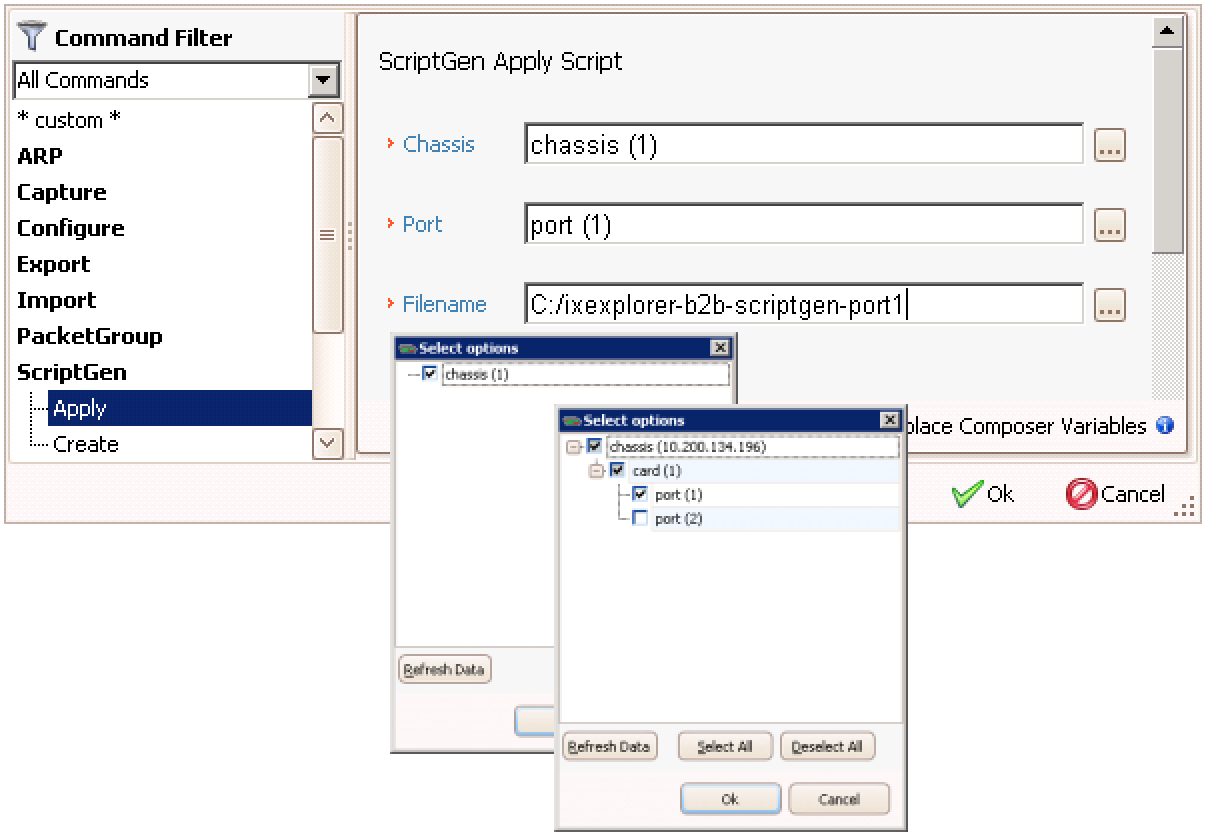


```
Welcome to tclsh.
% % source "C:\\Program Files\\Ixia\\Test Conductor

% source "C:\\Program Files\\Ixia\\Test Conductor\\

% Session Start  chassis=10.200.134.201 ports="(1.1
Starting IxExplorer session
```

Figure 35. Establishing an Ixia Traffic Generator session

13. Change the value of the **Command Type** to **Execute**.
14. Change the value of the **Session** column to *XM2*. Each Execute command must be associated with a unique session ID.
15. Use the **Command String** list to select the **ScriptGen Apply** command. This is required whenever an IxExplorer port configuration is to be loaded via Tcl.
16. Browse for the first of your chassis to populate the **Chassis** argument. **Chassis** is a required argument for this command.
17. Browse for the first of your chassis ports to populate the **Port** argument. **Port** is a required argument for this command.
18. Browse to populate the **Filename** argument. For example, browse to *C:/ixexplorer-b2b-scriptgen-port1.tcl*. The Tcl file will differ depending on the contents of the file that IxExplorer generated.

19. The **.tcl** file contains the port configuration for the first port. Click **Check** when done with line #2. This saves the values selected in the **Command** list wizard.



ScriptGen Apply Script

Chassis: chassis (1)

Port: port (1)

Filename: C:/ixexplorer-b2b-scriptgen-port1

Select options: chassis (1)

Select options: chassis (10.200.134.196) / card (1) / port (1) ✓, port (2)

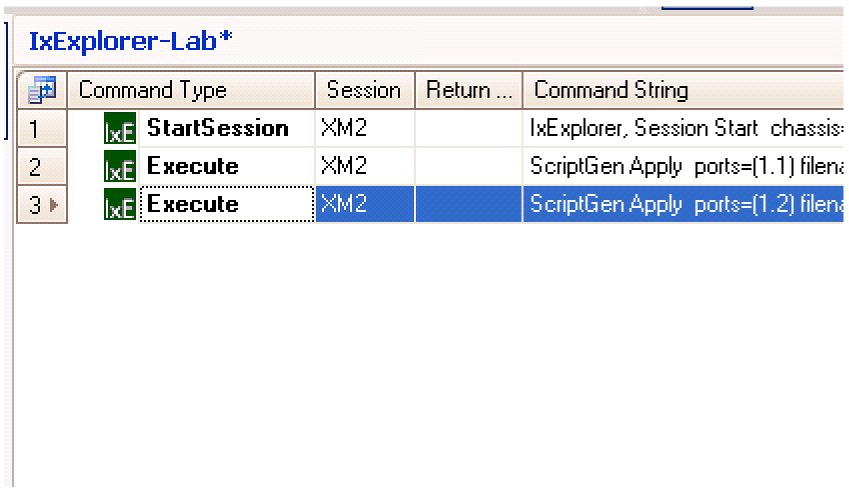
Figure 36. Creating a ScriptGen Apply command in the Composer editor

20. Highlight line #2. Click **Copy** and then click **Paste** to create line #3. This is required when configuration is performed on a per port basis and serves as a short cut to creating a command from scratch.

21. Click the **Command String** column to enter text edit mode. This allows you to enter text directly without needing to browse. Change the **Port** argument to your second chassis port. Similarly, change the **Filename** argument so that it points to your file; for example: *C:/ixexplorer-b2b-scriptgen-port2.tcl* as shown in Figure 36. This applies a different configuration file to a different port.

22. Press the **Enter** key to accept your text changes.

23. Click the **Save** icon to save your progress.



IxExplorer-Lab*

	Command Type	Session	Return ...	Command String
1	StartSession	XM2		IxExplorer, Session Start chassis
2	Execute	XM2		ScriptGen Apply ports=(1.1) filen
3	Execute	XM2		ScriptGen Apply ports=(1.2) filen

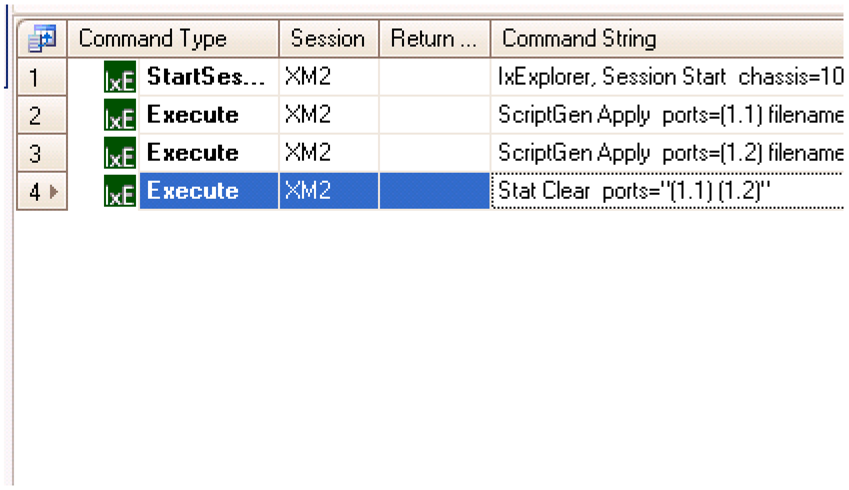
Figure 37. Copying a ScriptGen Apply command and editing arguments inline

24. Click **Insert Step Below** to create line #4.

25. Click the **Command String** list to browse for the **Stat Clear** command. This ensures that metric values are initialized when the test script executes.

26. Browse for both of your chassis ports to populate the **Port** argument. Port information is required for this command.

27. Click **Check** to save selections from the **Command** list wizard.



	Command Type	Session	Return ...	Command String
1	StartSes...	XM2		IxExplorer, Session Start chassis=10
2	Execute	XM2		ScriptGen Apply ports=(1.1) filename
3	Execute	XM2		ScriptGen Apply ports=(1.2) filename
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"

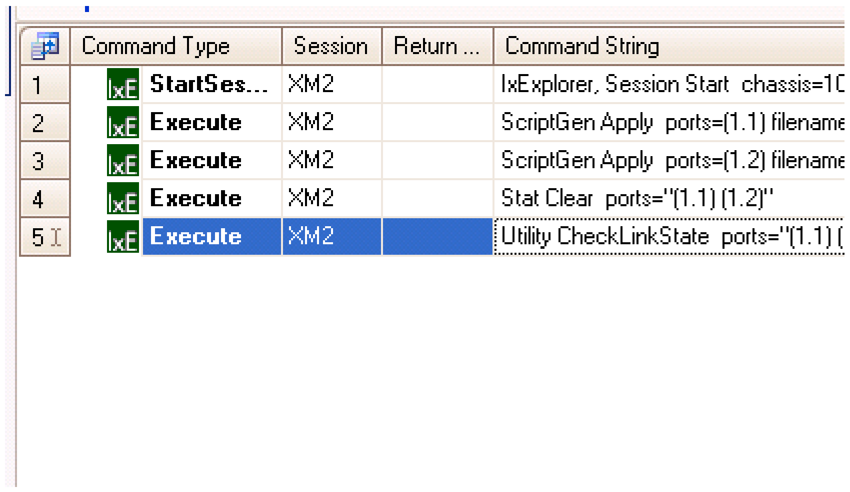
Figure 38. Inserting a Stat Clear command using the Composer editor

28. Click **Insert Step Below** to create line #5.

29. Click the **Command String** list to browse for the **Utility CheckLinkState** command. CheckLinkState is used to verify ports are up before proceeding.

30. Browse for both your chassis ports to populate the **Port** argument. Port information is a required argument for this command.

31. Click **Check** to accept.



	Command Type	Session	Return ...	Command String
1	StartSes...	XM2		IxExplorer, Session Start chassis=10
2	Execute	XM2		ScriptGen Apply ports=(1.1) filename
3	Execute	XM2		ScriptGen Apply ports=(1.2) filename
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.1) (

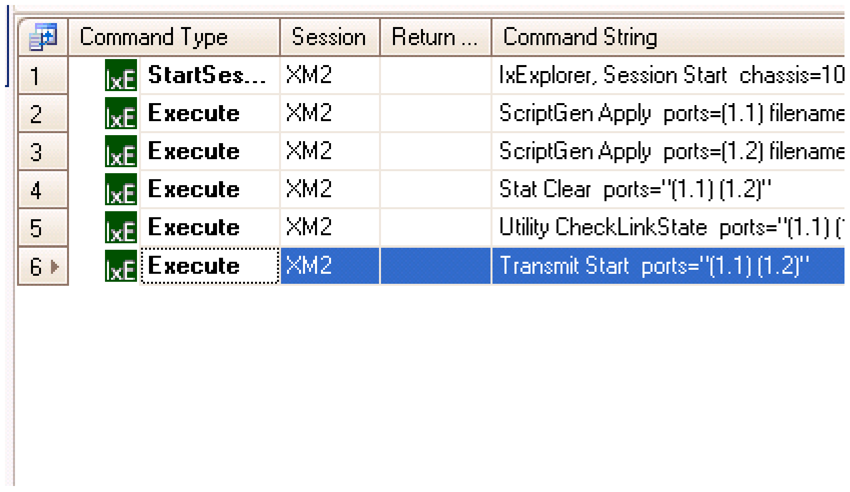
Figure 39. Inserting a Utility CheckLinkState command using the Composer editor

32. Click **Insert Step Below** to create line #6.

33. Click the **Command String** list to browse for the **Transmit Start** command. This includes a command to begin traffic streams.

34. Browse for both your chassis ports to populate the **Port** argument. Port information is a required argument for this command.

35. Click **Check** to accept.



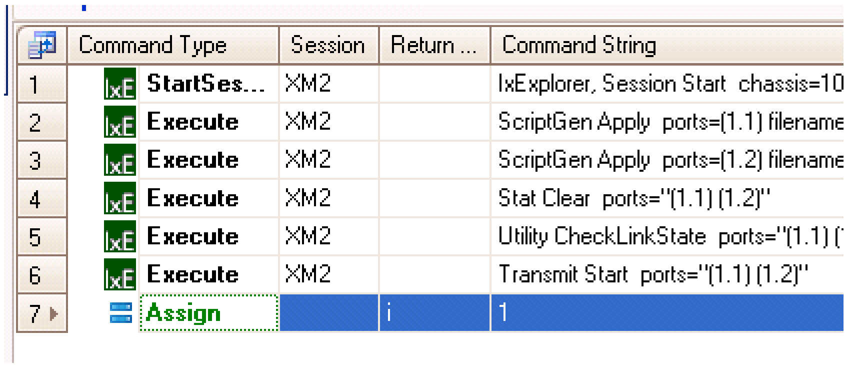
	Command Type	Session	Return ...	Command String
1	StartSes...	XM2		IxExplorer, Session Start chassis=10
2	Execute	XM2		ScriptGen Apply ports=(1.1) filename
3	Execute	XM2		ScriptGen Apply ports=(1.2) filename
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.1) (
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)"

Figure 40. Inserting a Transmit Start command using the Composer editor

36. Click **Insert Step Below** to create line #7.

37. Click the **Command Type** list to change the **Command Type** to **Assign**.

38. Click the **Command String** field to enter text
39. Enter the value *1* in the **Command String**. This assigns a value for the statement.
40. Click the **Return Variable** field to enter text.
41. Enter the value *i* for the variable. Its value is initialized to one.



	Command Type	Session	Return ...	Command String
1	StartSes...	XM2		IxExplorer, Session Start chassis=10
2	Execute	XM2		ScriptGen Apply ports=(1.1) filename
3	Execute	XM2		ScriptGen Apply ports=(1.2) filename
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.1) (
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)"
7	Assign		i	1

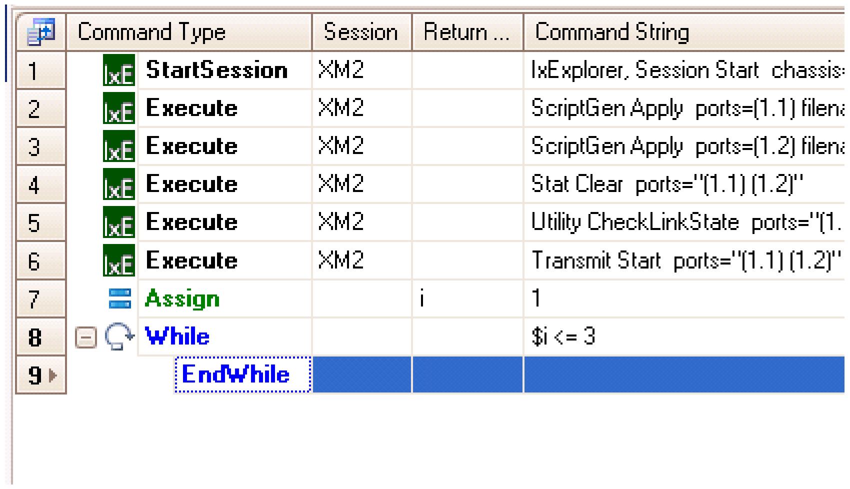
Figure 41. Assigning a numeric value to a variable using the Composer editor

42. Click **Insert Step Below** to create line #8.
43. Click the **Command Type** list to change the command type to **While**. This selects a flow control loop type. An **EndWhile** at line #9 will automatically be created to put a bound on the loop scope.
44. Click the **Command String** list to open the Tcl expression builder. This allows you to use the expression builder to assist in the creation of a loop expression.
45. Create the expression *$i* <= *3* to be evaluated during each iteration of the **While** loop. This step is required although the details of the expression will differ depending on the goal of the control flow being specified.

    You may type the expression directly into the lower evaluation window if you are familiar with Tcl syntax.

    If not, you may double click on the variables listed in the **Test Variables** window as well as the **Operators** list to help you build your expression. This step is recommended for users that are new to loop expressions.

46. Click **Check** to accept your changes.



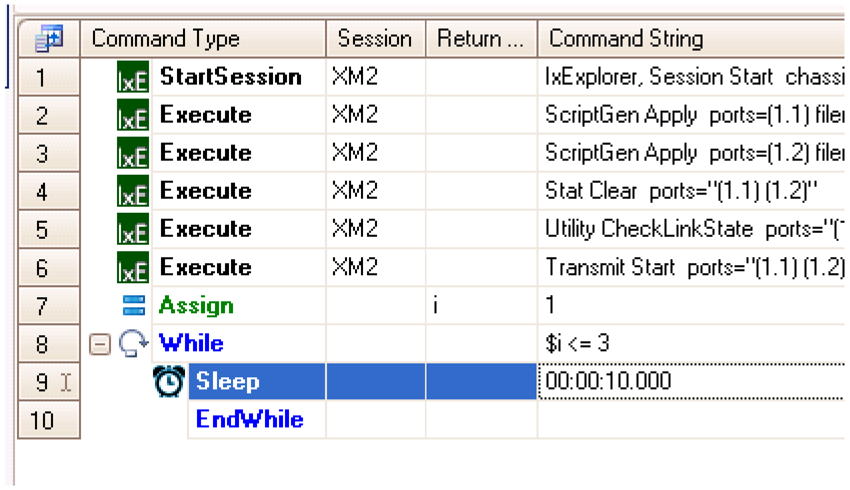
	Command Type	Session	Return ...	Command String
1	StartSession	XM2		IxExplorer, Session Start chassis
2	Execute	XM2		ScriptGen Apply ports=(1.1) filena
3	Execute	XM2		ScriptGen Apply ports=(1.2) filena
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)"
7	Assign		i	1
8	While			$i <= 3
9	EndWhile			

Figure 42. Iterating while a variable value is less than a maximum number

47. Highlight line #9. Click **Insert Step Above** to insert a new step at line #9.

48. Click the **Command Type** list to change the **Command Type** to **Sleep**. This allows traffic to continue for a fixed duration.

49. Edit the **Seconds** field of the **Command String** to the value of *10* to sleep for 10 seconds. The Sleep command should be indented inside the While/EndWhile lines.



	Command Type	Session	Return ...	Command String
1	StartSession	XM2		IxExplorer, Session Start chassi
2	Execute	XM2		ScriptGen Apply ports=(1.1) filei
3	Execute	XM2		ScriptGen Apply ports=(1.2) filei
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	EndWhile			

Figure 43. Inserting a Sleep command to paused for a fixed delay period

50. Highlight line #9. Click **Insert Step Below** to insert a new step at line #10.

51. Click the **Command Type** list to change the **Command Type** to **TclEval**. This step is required in order to have a way to make sure the While loop will exit.

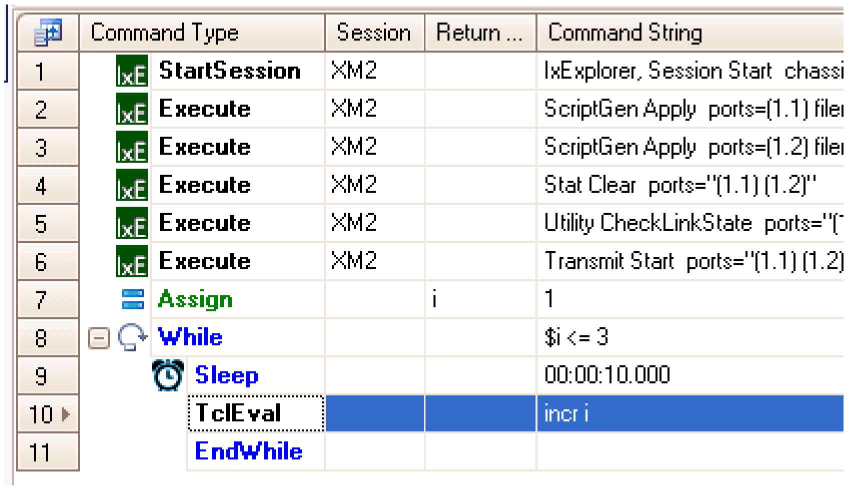
52. Click the **Command String** list to open the Tcl expression builder. This will provide assistance for building a loop expression.

53. Create the expression *incr i* to be evaluated as the last step of the **While** loop. This increments the loop index by one.

    You may type the expression directly into the bottom evaluation window if you are familiar with Tcl syntax.

    Otherwise you may double click on the variables listed in the **Test Variables** window or choose the **incr** operator from the **Available Commands** list under the **Variables** and **Procedures** heading in the **Command Category** list.

54. Click **Check** to accept your changes.



	Command Type	Session	Return ...	Command String
1	StartSession	XM2		IxExplorer, Session Start chassi
2	Execute	XM2		ScriptGen Apply ports=(1.1) filei
3	Execute	XM2		ScriptGen Apply ports=(1.2) filei
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			

Figure 44. Incrementing a Tcl variable using the Composer editor

55. Highlight line #11. Click **Insert Step Below** to insert a new step at Line #12.

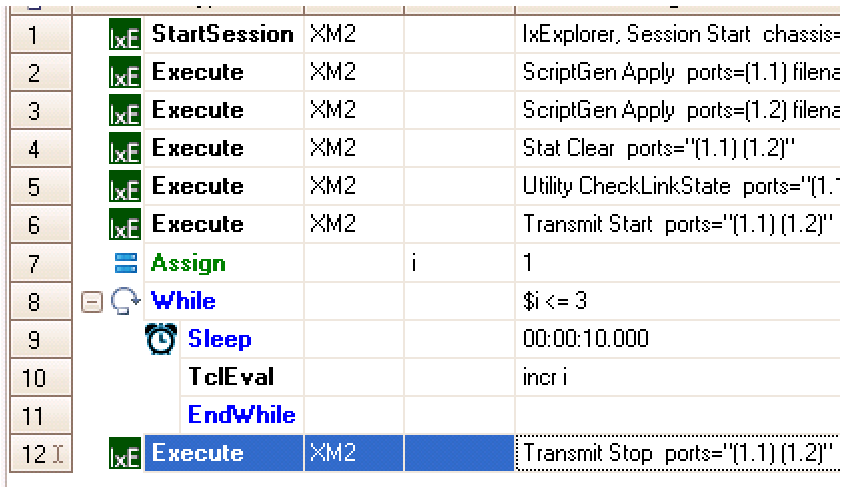
56. Change the value of the **Command Type** to **Execute**.

57. Change the value of the **Session** column to *XM2*. A session ID must match the command to be executed.

58. Click the **Command String** list to browse for the **Transmit Stop** command. This step is required to terminate the stream traffic.

59. Browse for the both of your chassis ports to populate the **Port** argument. Port information is a required argument for this command.

60. Click **Check** when done with line #12. This saves the changes made during edits in the **Command** list wizard.



1	StartSession	XM2		IxExplorer, Session Start chassis=
2	Execute	XM2		ScriptGen Apply ports=(1.1) filena
3	Execute	XM2		ScriptGen Apply ports=(1.2) filena
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.
6	Execute	XM2		Transmit Start ports="(1.1) (1.2)"
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1) (1.2)"

Figure 45. Inserting a Transmit Stop command using the Composer editor

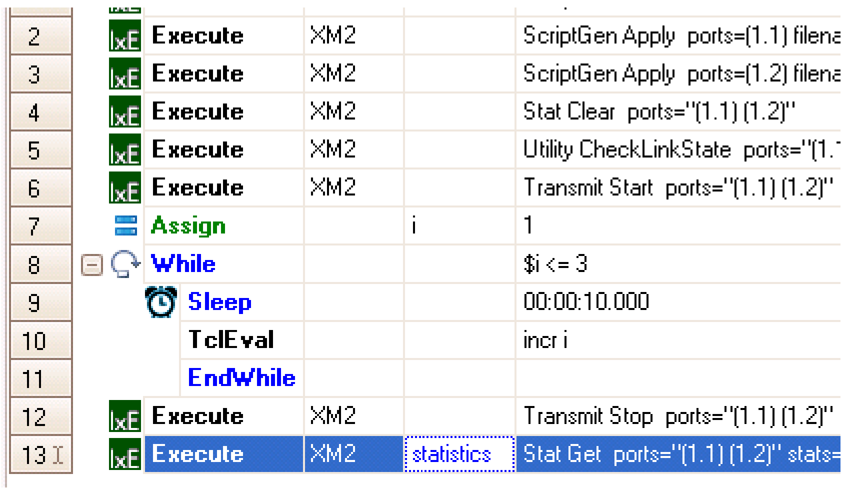
61. Click **Insert Step Below** to insert a new step at Line #13.

62. Click the **Command String** list to browse for the **Stat Get** command. This is used to collect one or more statistics from the count column in IxExplorer.

63. Browse for both of your chassis ports to populate the **Port** argument. Browse for the **framesReceived** and **framesSent** statistics to populate the **Stat** argument. These statistics are a common example of the types of statistics that are collected from an Ixia port. This is an example only, but these statistics will be required in later steps to create conditional statements.

64. Click **Check** when done with arguments for Line #13. This accepts changes made in the **Command** list wizard.

65. Enter the value *statistics* in the **Return Variable** column. This selects a container for the statistics returned by a **Stat Get** command. The variable **statistics** will be populated with the actual values from IxExplorer whenever the step is executed during **Playback** mode.



2	Execute	XM2		ScriptGen Apply ports={1.1} filena
3	Execute	XM2		ScriptGen Apply ports={1.2} filena
4	Execute	XM2		Stat Clear ports="{1.1} {1.2}"
5	Execute	XM2		Utility CheckLinkState ports="{1.
6	Execute	XM2		Transmit Start ports="{1.1} {1.2}"
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="{1.1} {1.2}"
13	Execute	XM2	statistics	Stat Get ports="{1.1} {1.2}" stats=

Figure 46. Inserting a Stat Get command with return variable using Composer editor

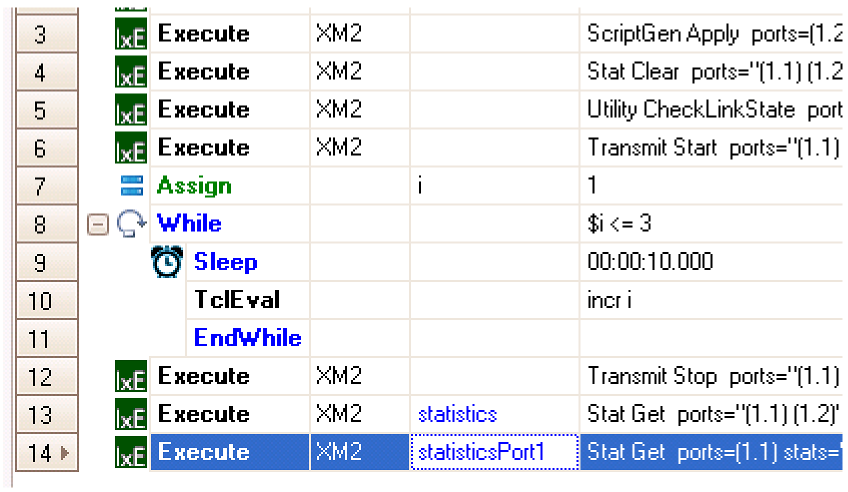
66. Click **Insert Step Below** to insert a new step at line #14.

67. Click the **Command String** list to browse for the **Stat Get** command. Browse for only the first of your chassis ports to populate the **Port** argument. This is a convenient that you can used to collect statistics for a single port or group of ports.

68. Browse for the **framesReceived** and **framesSent** statistics to populate the **Stat** argument. A choice of statistics is a required argument for this command.

69. Click **Check** when done with arguments for line #14. This accepts changes made in the **Command** list wizard.

70. Enter the value *statisticsPort1* for the **Return Variable** column. This establishes the variable that will hold the collected. The variable **statisticsPort1** is a convenient variable for storing the results associated with just the first chassis port.



3	Execute	XM2		ScriptGen Apply ports=(1.2
4	Execute	XM2		Stat Clear ports="(1.1) (1.2
5	Execute	XM2		Utility CheckLinkState port
6	Execute	XM2		Transmit Start ports="(1.1)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1)
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)'
14	Execute	XM2	statisticsPort1	Stat Get ports=(1.1) stats='

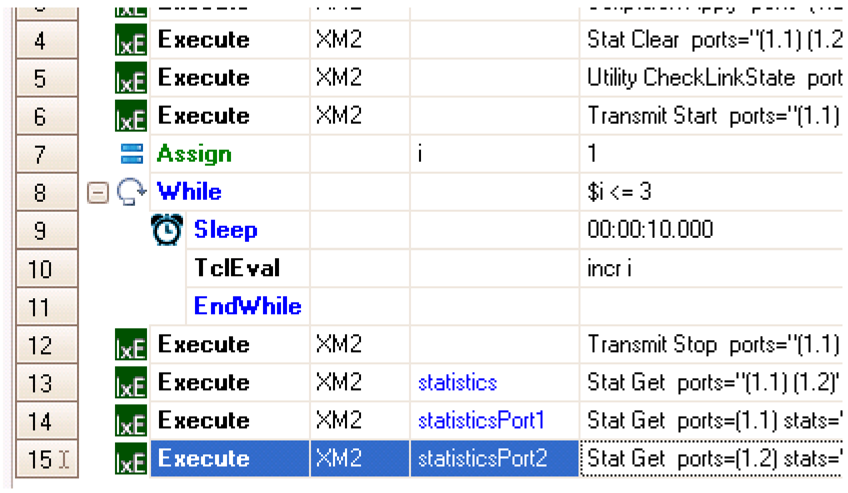
Figure 47. Inserting a Stat Get command for a single port using Composer editor

71. Click **Copy** and then **Paste** to insert a new step at line #15.

72. Edit the **Command String** text to change the **Port** argument to the second chassis port. This allows this command to collect stats on a different port.

73. Edit the value of the **Return Variable** column to *statisticsPort2*. This creates a different variable to contain the statistics for the second port.

The variable **statisticsPort2** is a convenience for storing only the results associated with just the second chassis port.



4	Execute	XM2		Stat Clear ports="(1.1) (1.2
5	Execute	XM2		Utility CheckLinkState port
6	Execute	XM2		Transmit Start ports="(1.1)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1)
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)'
14	Execute	XM2	statisticsPort1	Stat Get ports=(1.1) stats='
15	Execute	XM2	statisticsPort2	Stat Get ports=(1.2) stats='

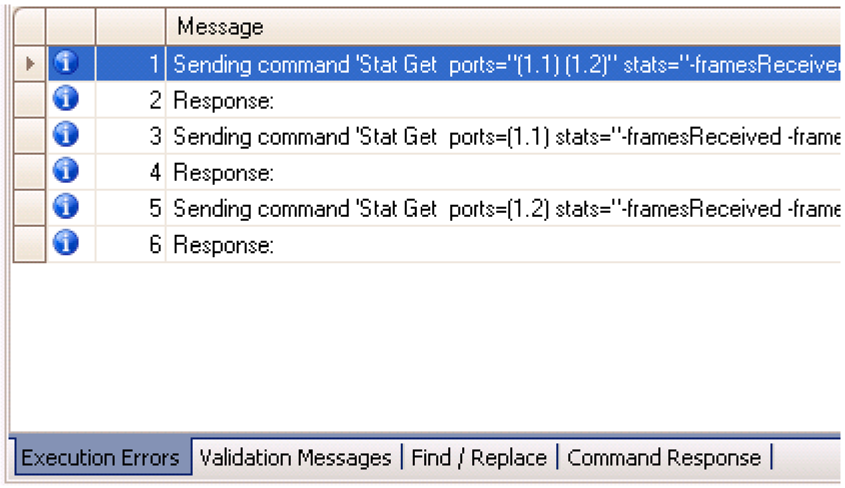
Figure 48. Copying a Stat Get command and modifying arguments inline

74. Highlight line #13 through line #15. This allows multiple lines to be replayed. Click **Play** to send the **Stat Get** commands to the Ixia chassis, which will populate information in the variables you created. Container variables will not hold any values or variable structure unless the **Stat Get** commands are issued within an active session.

    As the commands execute, status messages will appear in the **Execution Errors** window.

75. Remember to click **Save** periodically to save your work. This step is strongly recommended so that work is captured on a regular basis in case there is a need to go back to an earlier version.

76. Return variables that have not been fully populated will be colored **red**. You will see corresponding error messages in the **Validation Messages** window.



			Message
▸	●	1	Sending command 'Stat Get ports="(1.1) (1.2)" stats="-framesReceive
	●	2	Response:
	●	3	Sending command 'Stat Get ports=(1.1) stats="-framesReceived -frame
	●	4	Response:
	●	5	Sending command 'Stat Get ports=(1.2) stats="-framesReceived -frame
	●	6	Response:

Figure 49. Logging the executing of stat collection commands in Execution Errors tab

77. Click **Insert Step Below** to create line #16.

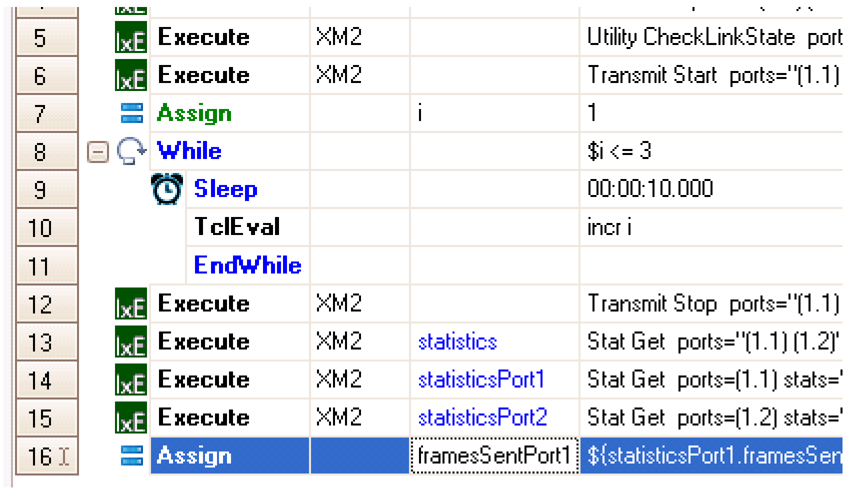
78. Click the **Command Type** list to change the **Command Type** to **Assign**. This is required as the previous command was of a different type.

79. Click the **Command String** list to open the Tcl expression builder. This is used for assistance in building a Tcl expression.

80. Create the expression *${statisticsPort1.framesSent}* by expanding the **statisticsPort1** tree item under **Test Variables** and double-clicking the **framesSent** statistic. This provides you the correct variable syntax.

81. Click **Check** to accept your changes.

82. (Optional) Enter *framesSentPort1* for the **Return Variable**. This defines a shorter name for the variable.



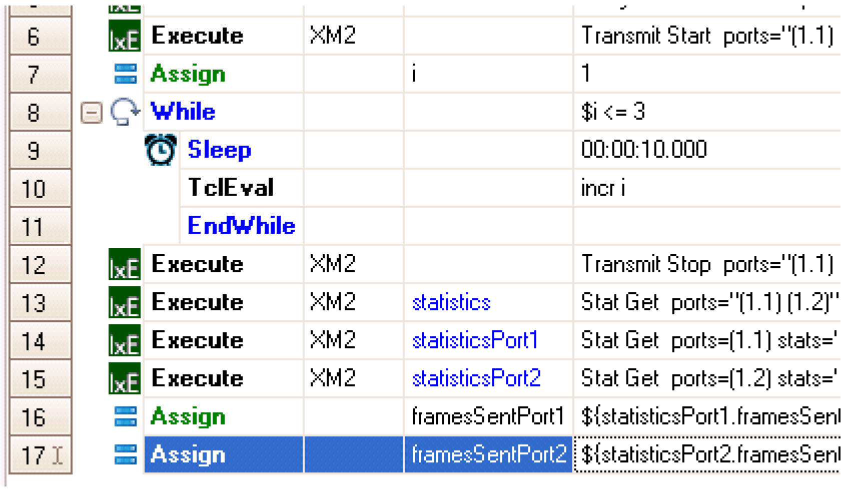
5	Execute	XM2		Utility CheckLinkState port
6	Execute	XM2		Transmit Start ports="(1.1)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1)
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)'
14	Execute	XM2	statisticsPort1	Stat Get ports=(1.1) stats='
15	Execute	XM2	statisticsPort2	Stat Get ports=(1.2) stats='
16	Assign		framesSentPort1	${statisticsPort1.framesSen

Figure 50. Assigning a composite variable statistic to a new variable using Composer editor

83. Click **Copy** and then **Paste** to insert a new step at line #17.

84. Edit the Command String text to change the expression to *${statisticsPort2.framesSent}* for the second chassis port. This is required in order to collect the same statistic from a different port.

85. (Optional) Edit the value of the **Return Variable** column to *framesSentPort2*. This defines a shorter name for the variable.



6	Execute	XM2		Transmit Start ports="(1.1)
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1)
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)"
14	Execute	XM2	statisticsPort1	Stat Get ports=(1.1) stats='
15	Execute	XM2	statisticsPort2	Stat Get ports=(1.2) stats='
16	Assign		framesSentPort1	${statisticsPort1.framesSent
17	Assign		framesSentPort2	${statisticsPort2.framesSent

Figure 51. Copying a variable assignment and modifying arguments in line

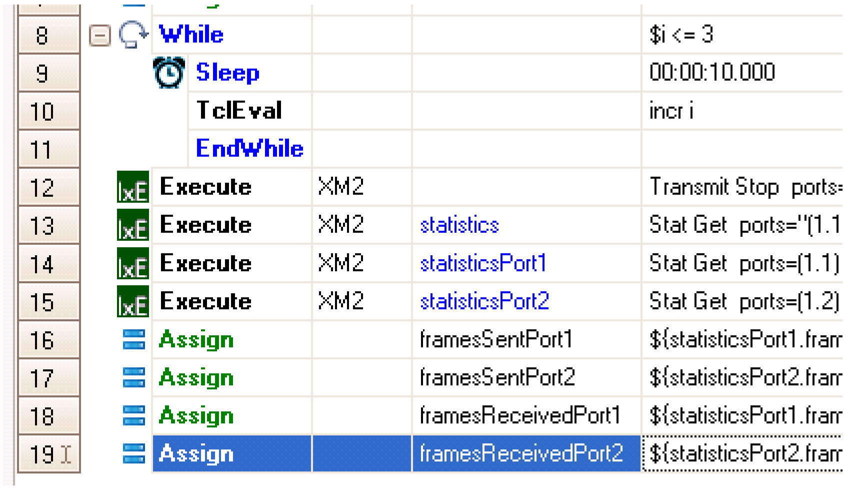
86. Highlight line #16 and line #17. Click **Copy**, highlight line #17 again, and then click **Paste** to create line #18 and line #19. This step is required in order to paste a duplicate copy of multiple lines into the same location.

87. Edit Command String for Line #18 to change its value to *${statisticsPort1.framesReceived}*.

88. Edit **Return Variable** for line #18 to change its value to *framesReceivedPort1*. This defines a shorter name for a statistic.

89. Edit Command String for line #19 to change its value to *${statisticsPort2.framesReceived}*. This accesses a different stat on the second port.

90. Edit **Return Variable** for line #19 to change its value to *framesReceivedPort2*.



8	While			$i <= 3
9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports=
13	Execute	XM2	statistics	Stat Get ports="(1.1
14	Execute	XM2	statisticsPort1	Stat Get ports=(1.1)
15	Execute	XM2	statisticsPort2	Stat Get ports=(1.2)
16	Assign		framesSentPort1	${statisticsPort1.fram
17	Assign		framesSentPort2	${statisticsPort2.fram
18	Assign		framesReceivedPort1	${statisticsPort1.fram
19	Assign		framesReceivedPort2	${statisticsPort2.fram

Figure 52. Copying multiple Assign statements at once and editing arguments of second copies

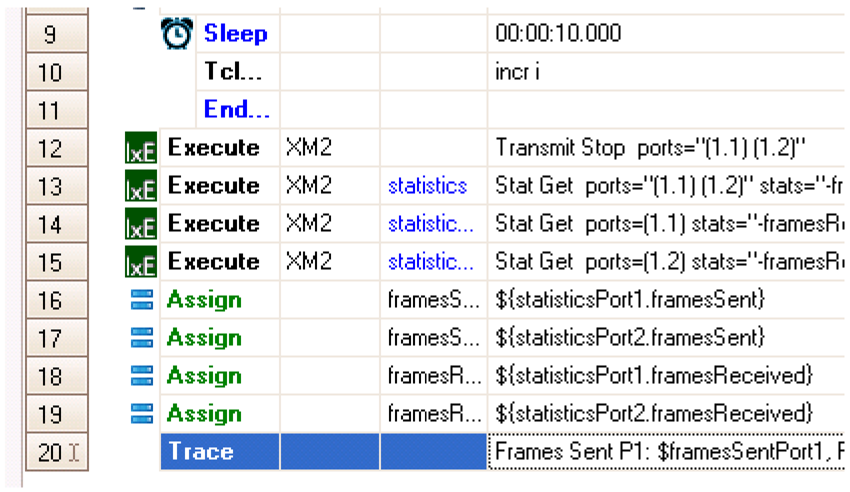
91. Click **Insert Step Below** to create line #20.

92. Click the **Command Type** list to change the **Command Type** to **Trace**. The last command inserted was of a different type.

93. Click the **Command String** list to open the Tcl expression builder.

94. Create the expression *Frames Sent P1: $framesSentPort1, Frames Received P1: $framesReceivedPort1* by selecting the appropriate items under **Test Variables** and adding the additional text needed for the debugging output message. The Trace must have an argument string, although the argument string may contain any combination of variables and text required for the message to be output.

95. Click **Check** to accept your changes. This saves changes made in the **Command** list wizard.

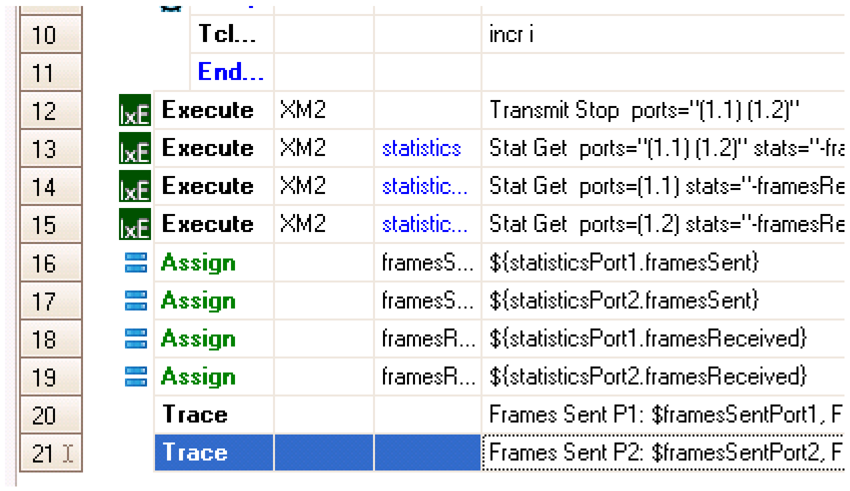


9	Sleep			00:00:10.000
10	Tcl...			incr i
11	End...			
12	Execute	XM2		Transmit Stop ports="{1.1} {1.2}"
13	Execute	XM2	statistics	Stat Get ports="{1.1} {1.2}" stats="-fr
14	Execute	XM2	statistic...	Stat Get ports={1.1} stats="-framesR
15	Execute	XM2	statistic...	Stat Get ports={1.2} stats="-framesR
16	Assign		framesS...	${statisticsPort1.framesSent}
17	Assign		framesS...	${statisticsPort2.framesSent}
18	Assign		framesR...	${statisticsPort1.framesReceived}
19	Assign		framesR...	${statisticsPort2.framesReceived}
20	Trace			Frames Sent P1: $framesSentPort1, F

Figure 53. Inserting a debugging statement using Composer editor

96. Click **Copy** and then **Paste** to insert a new step at line #21.

97. Edit the Command String text to change the expression to reference the second port *Frames Sent P2: $framesSentPort2, Frames Received P2: $framesReceivedPort2*. This changes the output message to refer to values on the second port.



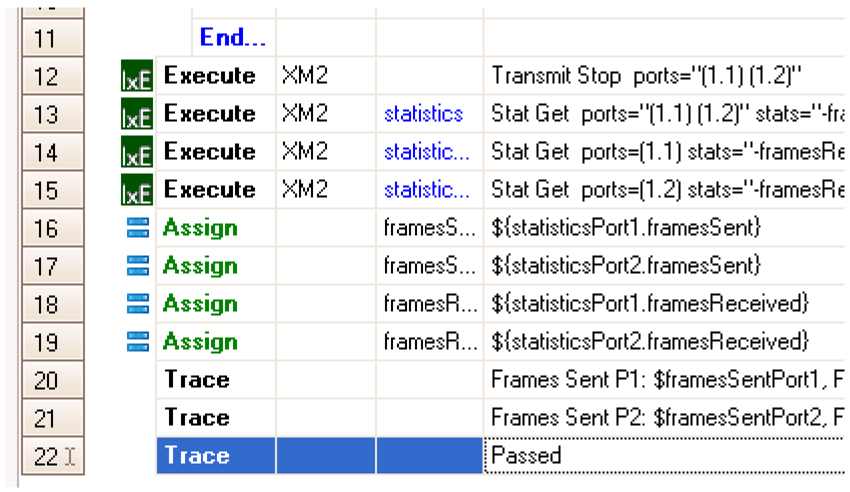
10	Tcl...			incr i
11	End...			
12	Execute	XM2		Transmit Stop ports="{1.1} {1.2}"
13	Execute	XM2	statistics	Stat Get ports="{1.1} {1.2}" stats="-fra
14	Execute	XM2	statistic...	Stat Get ports={1.1} stats="-framesRe
15	Execute	XM2	statistic...	Stat Get ports={1.2} stats="-framesRe
16	Assign		framesS...	${statisticsPort1.framesSent}
17	Assign		framesS...	${statisticsPort2.framesSent}
18	Assign		framesR...	${statisticsPort1.framesReceived}
19	Assign		framesR...	${statisticsPort2.framesReceived}
20	Trace			Frames Sent P1: $framesSentPort1, F
21	Trace			Frames Sent P2: $framesSentPort2, F

Figure 54. Copying a debugging statement and modifying arguments in Composer editor

98. Click **Insert Step Below** to create line #22.

99. Click the **Command Type** list to change the **Command Type** to **Trace**.

100. Edit the **Command String** text to *Passed* – this will be output as a debugging message. This step is a convenience for the test reader, so that there is a clear message displayed when a test is successful.



11	End...			
12	Execute	XM2		Transmit Stop ports="(1.1) (1.2)"
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)" stats="-fra
14	Execute	XM2	statistic...	Stat Get ports=(1.1) stats="-framesRe
15	Execute	XM2	statistic...	Stat Get ports=(1.2) stats="-framesRe
16	Assign		framesS...	${statisticsPort1.framesSent}
17	Assign		framesS...	${statisticsPort2.framesSent}
18	Assign		framesR...	${statisticsPort1.framesReceived}
19	Assign		framesR...	${statisticsPort2.framesReceived}
20	Trace			Frames Sent P1: $framesSentPort1, F
21	Trace			Frames Sent P2: $framesSentPort2, F
22	Trace			Passed

Figure 55. Inserting a debugging statement using Composer editor

101. Click **Files Catalog** and then click **Add File** to create a new row. This is needed when the test will access an input or output file. The file will be attached to the composer script.

102. Change the **Type** to **Result**. Enter *C:/IxExplorer.csv* for the **File** location. This output file is an example and can be changed to reflect the specific test executed.

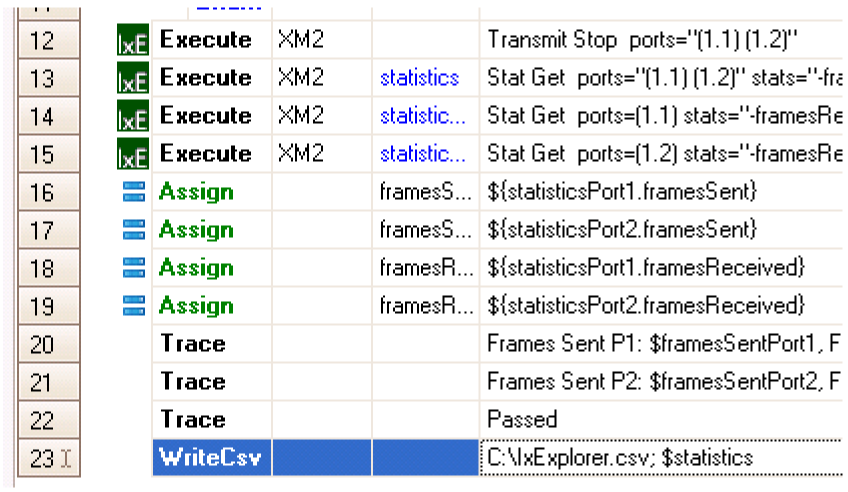
103. Click **OK** to accept **file1** as the **Composer Variable**. This step accepts the default file name, although it may be changed to a more meaningful name if desired. From this point on the file will be referred to by its variable, rather than its absolute path.

104. Click **Insert Step Below** to create line #23. Click the **Command Type** list to change the **Command Type** to **WriteCSV**. This step provides an automatic way of generating comma separated values in an output file, consisting of variables created by the composer script.

105. Click the **Command String** list to open the argument builder. This provides assistance with the variables that are available to be output by the composer script at this line in the script.

106. Click **Select Variable** and check **Use Variable** to specify *file1* as the **File Name**. Enter *$statistics* as the **Variable List** argument. The file name and one or more variables are required arguments of this command.

107. Click **Check** to save changes made in the **Command** list wizard.

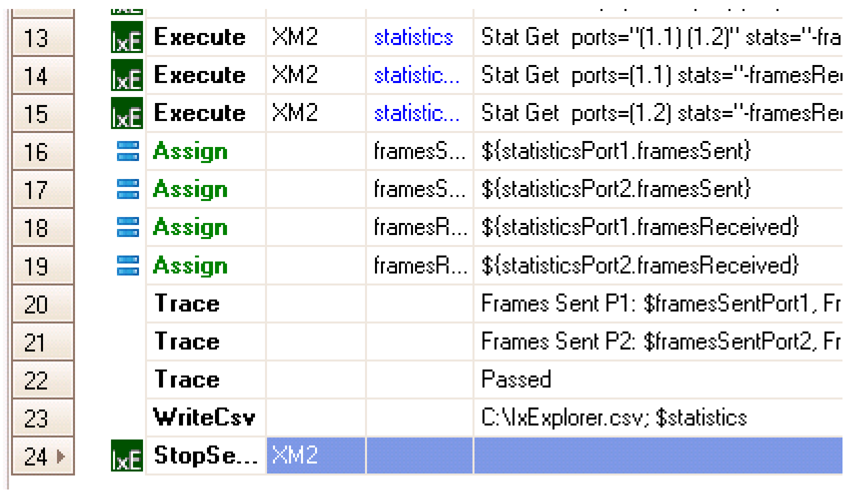


12	Execute	XM2		Transmit Stop ports="(1.1) (1.2)"
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)" stats="-fra
14	Execute	XM2	statistic...	Stat Get ports=(1.1) stats="-framesRe
15	Execute	XM2	statistic...	Stat Get ports=(1.2) stats="-framesRe
16	Assign		framesS...	${statisticsPort1.framesSent}
17	Assign		framesS...	${statisticsPort2.framesSent}
18	Assign		framesR...	${statisticsPort1.framesReceived}
19	Assign		framesR...	${statisticsPort2.framesReceived}
20	Trace			Frames Sent P1: $framesSentPort1, F
21	Trace			Frames Sent P2: $framesSentPort2, F
22	Trace			Passed
23	WriteCsv			C:\IxExplorer.csv; $statistics

Figure 56. Inserting a WriteCSV command to output statistics using Composer editor

108. Click **Disconnect** to close the IxExplorer session, which will automatically create line #24 with the **Command Type** of **StopSession**. This terminates a session while generating a stop command.

109. Make sure to click the **Save** icon to save this version of the IxExplorer script before proceeding. This ensures that the composer script is syntactically valid before proceding to the next round of debugging.



13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)" stats="-fra
14	Execute	XM2	statistic...	Stat Get ports=(1.1) stats="-framesRe
15	Execute	XM2	statistic...	Stat Get ports=(1.2) stats="-framesRe
16	Assign		framesS...	${statisticsPort1.framesSent}
17	Assign		framesS...	${statisticsPort2.framesSent}
18	Assign		framesR...	${statisticsPort1.framesReceived}
19	Assign		framesR...	${statisticsPort2.framesReceived}
20	Trace			Frames Sent P1: $framesSentPort1, Fr
21	Trace			Frames Sent P2: $framesSentPort2, Fr
22	Trace			Passed
23	WriteCsv			C:\IxExplorer.csv; $statistics
24	StopSe...	XM2		

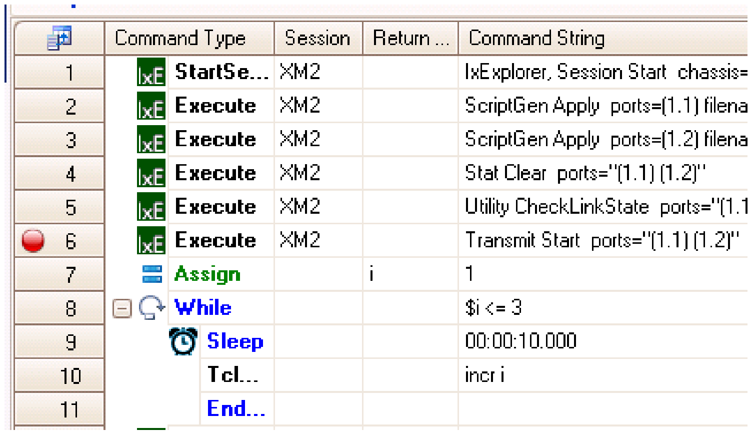
Figure 57. Inserting a StopSession command automatically upon session disconnect

110. Click **Debug** to switch to the Debugger.

111. (Optional) Click the **Line #** column next to line #6 and line #22 to add a breakpoint that will pause execution at each of those lines. This marks clear exit points in the composer script that are expected to be reached during execution.

You may add additional breakpoints to other lines where you wish to pause the execution of your script.

If you wish to disable a breakpoint, simply click on the **Breakpoint** icon. The icon will appear as an empty circle, indicating that the breakpoint is present but disabled.



	Command Type	Session	Return ...	Command String
1	StartSe...	XM2		IxExplorer, Session Start chassis=
2	Execute	XM2		ScriptGen Apply ports=(1.1) filena
3	Execute	XM2		ScriptGen Apply ports=(1.2) filena
4	Execute	XM2		Stat Clear ports="(1.1) (1.2)"
5	Execute	XM2		Utility CheckLinkState ports="(1.1
● 6	Execute	XM2		Transmit Start ports="(1.1) (1.2)"
7	Assign		i	1
8	While			$i <= 3
9	Sleep			00:00:10.000
10	Tcl...			incr i
11	End...			

Figure 58. Inserting a breakpoint to pause execution of a command in Composer debugger

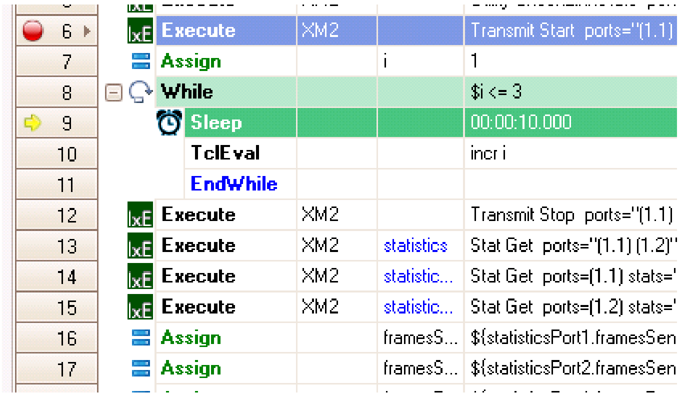
112. Click **Play** to start execution of the test. The test will execute until the first breakpoint is reached.

113. Click **Play** again to continue execution to the next breakpoint and observe the effect in IxExplorer where appropriate.

114. Make sure that you are able to reach the breakpoint associated with the **Passed-Trace** message on line #22 before continuing further. This step validates that the execution of the script in this example was successful. If you can successfully reach this breakpoint then your script is working as expected.

115. Click **Play** to execute to the end of the script.

116. You may use the **Global Output** window to assist you in verifying the correct execution of your script. Use this window for monitoring the real-time progress of a composer script and viewing output messages from all sessions in one place.



● 6	Execute	XM2		Transmit Start ports="(1.1)
7	Assign		i	1
8	While			$i <= 3
⇨ 9	Sleep			00:00:10.000
10	TclEval			incr i
11	EndWhile			
12	Execute	XM2		Transmit Stop ports="(1.1)
13	Execute	XM2	statistics	Stat Get ports="(1.1) (1.2)"
14	Execute	XM2	statistic...	Stat Get ports=(1.1) stats='
15	Execute	XM2	statistic...	Stat Get ports=(1.2) stats='
16	Assign		framesS...	${statisticsPort1.framesSen
17	Assign		framesS...	${statisticsPort2.framesSen

Figure 59. Monitoring the execution of commands in the Composer debugger

Test Case: Leveraging Test Automation in a Vendor Selection Process

Overview

A key requirement of an IP equipment vendor selection process is that the criteria applied are consistent across individual test runs as well as across vendor implementations. An organization seeking to select from a set of potential equipment vendors will be able to do so with confidence by leveraging a consistent and repeatable automation scheme,

Objective

The goal of this methodology is to develop a repeatable process for the collection of the key IP metrics needed to choose between multiple vendor offerings. The Ixia Test Composer test authoring tool will be used to generate vendor specific procedures for configuring the device under test (DUT) from each equipment manufacturer.

Setup

The script generated by Composer will be organized into high-level methods that correspond to best practices. Traffic is injected into the DUT and statistics are retrieved for automated success/failure analysis. The graphical script will be used by the Test Conductor Regression Manager and Scheduler to runs the vendor selection process multiple times. Vendor A, B, and C are to be evaluated using automation to guarantee fair, accurate, and repeatable results.

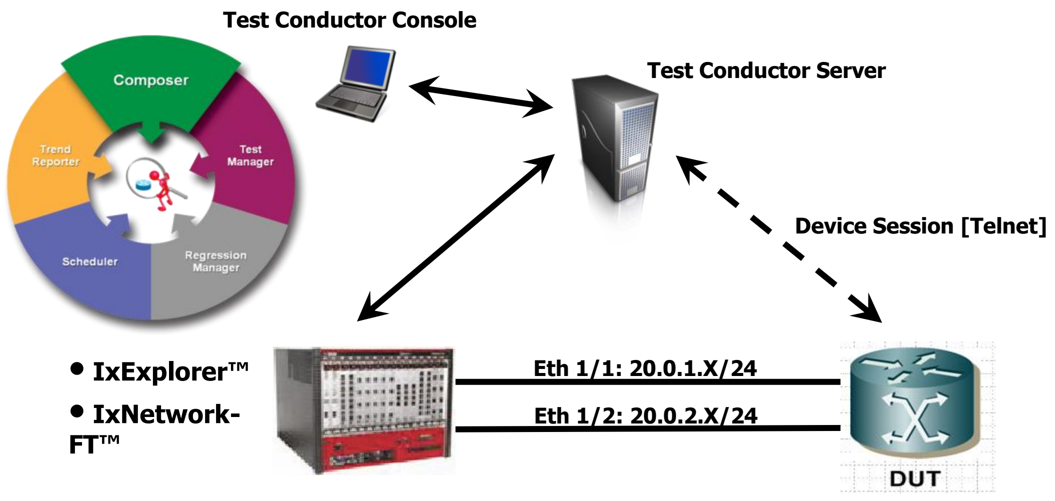


Figure 60. Vendor Selection Composer Methodology Logical Diagram

Step-by-Step Instructions

1. Use the **Composer Session Manager** to log in to Vendor A's device at the enable/administrator level. Configure layer 2-3 and/or layer 4-7 interface properties such as IP Address, VLANs, etc., to bring Vendor A's test interfaces up to operational level.

2. Issue Vendor A's device status commands to verify the operational status of the DUT in preparation for its participation in the vendor selection process. The **Composer** will capture the commands issued to Vendor A's device, along with the status responses to those commands.

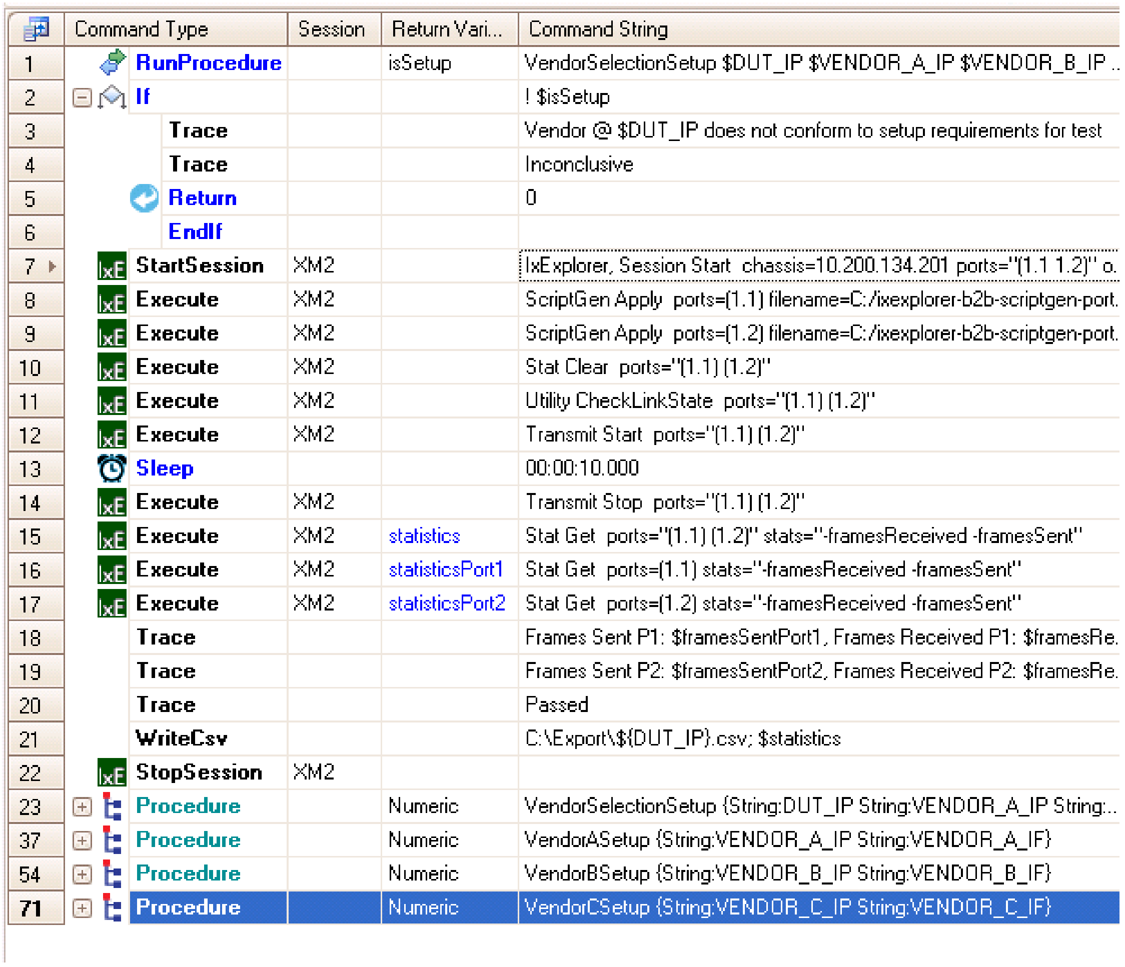
3. For future automation purposes, identify the key values contained in the status messages and use the **Composer Command Response Map Editor** to mark these values as being associated with variables. These variables may be inspected during the execution of the **Composer** script to test for correctness against the initial device setup.

4. Use **Composer Control Flow Logic** statements such as **If**, **While**, **For**, etc. to test the state of the response variables. Test for valid start values. Start with the configuration of a small number of interfaces before scaling up to the number of interfaces needed by the test traffic portion of the vendor selection process.

5. Use the **Composer Procedure Block** to encapsulate the captured statements, responses, variables and control flow logic into a reusable unit of device configuration for Vendor A. Modify the DUT setup procedure body so that hard-coded values such as IP Addresses, VLANS, etc. are variables recognized by the **Composer**.

6. Create **Input Parameters** with the same name as the variables you identified, so that the DUT setup procedures can be executed against other Vendor A family devices that support the same syntax, but may have different configuration values - such as test interface IP Addresses.

7. Identify an output variable to indicate the overall success or failure of the Vendor A setup process. This variable could be a zero/one for failure/success. It should be checked in the vendor selection process script's main body. If the vendor device has not been successfully configured to an operational state needed by the selection process, then the script needs to mark the vendor device as non-conformant to setup criteria. The script should mark its overall state as inconclusive, as no valid testing determination can be made.
8. Use the **Composer** to create a Vendor B setup procedure with input parameters and internal variables consistent with the Vendor A setup procedure. The configuration syntax could be very different in structure, but since the two vendors are competing for the same feature there should be enough commonality to identify the corresponding variable settings on each vendor device. If there are differences in structure these can be handled through input arguments, some of which can be ignored for one vendor or another.



	Command Type	Session	Return Vari...	Command String
1	RunProcedure		isSetup	VendorSelectionSetup $DUT_IP $VENDOR_A_IP $VENDOR_B_IP ..
2	If			! $isSetup
3	Trace			Vendor @ $DUT_IP does not conform to setup requirements for test
4	Trace			Inconclusive
5	Return			0
6	EndIf			
7	StartSession	XM2		IxExplorer, Session Start chassis=10.200.134.201 ports="{1.1 1.2}" o.
8	Execute	XM2		ScriptGen Apply ports={1.1} filename=C:/ixexplorer-b2b-scriptgen-port.
9	Execute	XM2		ScriptGen Apply ports={1.2} filename=C:/ixexplorer-b2b-scriptgen-port.
10	Execute	XM2		Stat Clear ports="{1.1} {1.2}"
11	Execute	XM2		Utility CheckLinkState ports="{1.1} {1.2}"
12	Execute	XM2		Transmit Start ports="{1.1} {1.2}"
13	Sleep			00:00:10.000
14	Execute	XM2		Transmit Stop ports="{1.1} {1.2}"
15	Execute	XM2	statistics	Stat Get ports="{1.1} {1.2}" stats="-framesReceived -framesSent"
16	Execute	XM2	statisticsPort1	Stat Get ports={1.1} stats="-framesReceived -framesSent"
17	Execute	XM2	statisticsPort2	Stat Get ports={1.2} stats="-framesReceived -framesSent"
18	Trace			Frames Sent P1: $framesSentPort1, Frames Received P1: $framesRe.
19	Trace			Frames Sent P2: $framesSentPort2, Frames Received P2: $framesRe.
20	Trace			Passed
21	WriteCsv			C:\Export\${DUT_IP}.csv; $statistics
22	StopSession	XM2		
23	Procedure		Numeric	VendorSelectionSetup {String:DUT_IP String:VENDOR_A_IP String:..
37	Procedure		Numeric	VendorASetup {String:VENDOR_A_IP String:VENDOR_A_IF}
54	Procedure		Numeric	VendorBSetup {String:VENDOR_B_IP String:VENDOR_B_IF}
71	Procedure		Numeric	VendorCSetup {String:VENDOR_C_IP String:VENDOR_C_IF}

Figure 61. Vendor device configuration organized into high level procedures with arguments

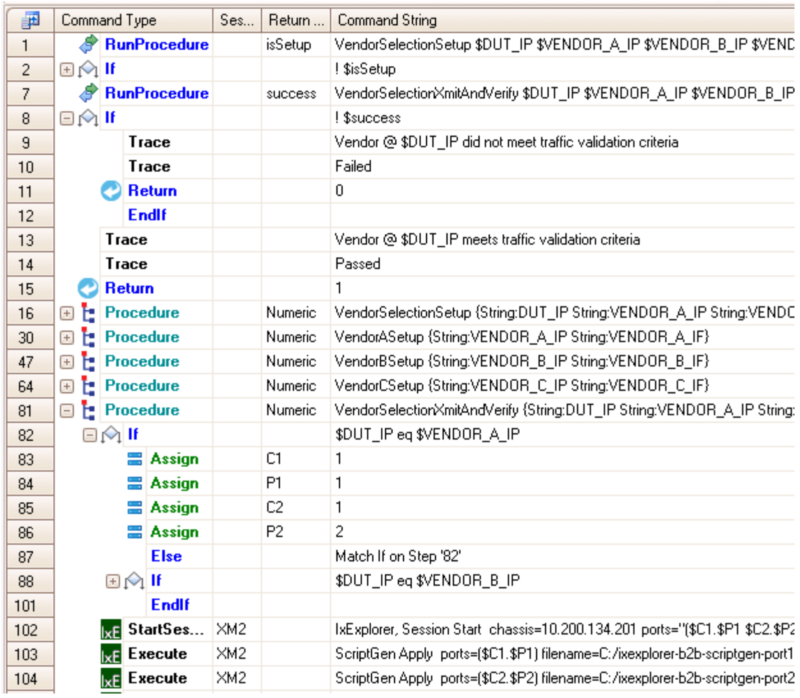
Think of this as an evolving process in which you are going to start with the vendor-specific details of each device so as to initialize the DUT into the state needed by the vendor selection process, using vendor-specific syntax. As you introduce other device vendors into the vendor selection script, you may need to modify your assumptions about how a vendor decided to implement a feature. Through this process you will discover the next level of DUT setup abstraction.

9. Use the **Composer** to capture the configuration commands of Vendor C and parameterize the input variables to be as consistent as possible with those of Vendor A and Vendor B. Make sure to return a clear success or failure from the setup procedure as in the other two cases, so that any vendors who fail to be conformant with the expected environment can be identified at the start of the process, not the end.

10. Use what you have learned about the structure of the three Vendors to create a generic **VendorSelectionSetup** procedure which corresponds to the generic interface you expect all vendors to provide, regardless of the specific details of their syntax. This high-level method

can be implemented as a procedure that switches to the appropriate low level vendor setup procedure based upon a selector such as device IP Address, for example.

11. Use the **Composer** to capture the Ixia traffic generator commands needed to inject the vendor selection traffic pattern into the DUT. This corresponds to the main body of your vendor selection test. **Composer** makes it possible to take a saved traffic generator port configuration and apply it to a different set of ports. Just as was done for the DUT setup, change the appropriate hard-coded values such as chassis card and port to parameters that can be selected to match the current vendor under test by a selector such as DUT IP Address.



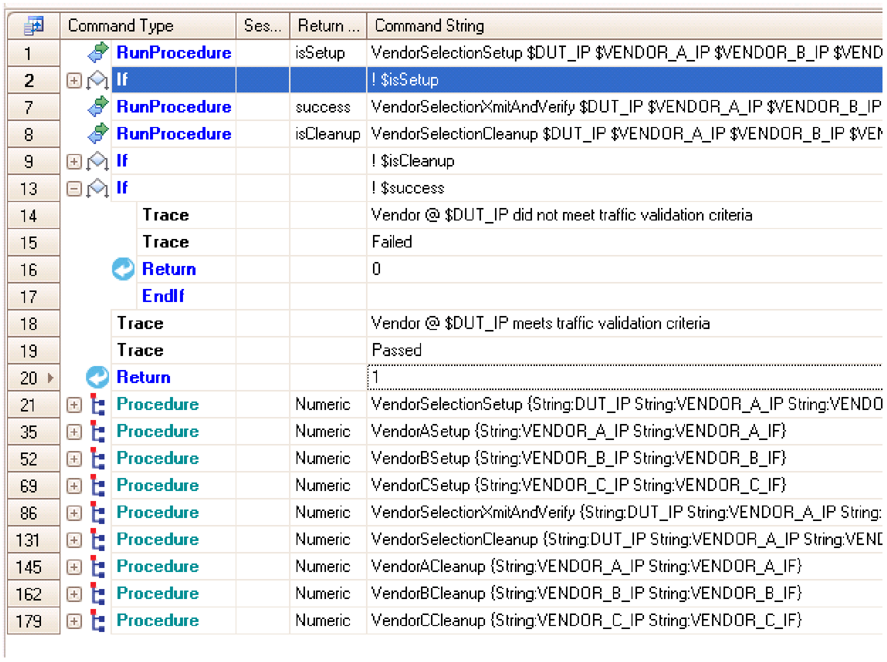
	Command Type	Ses...	Return ...	Command String
1	RunProcedure		isSetup	VendorSelectionSetup $DUT_IP $VENDOR_A_IP $VENDOR_B_IP $VEND
2	If			! $isSetup
7	RunProcedure		success	VendorSelectionXmitAndVerify $DUT_IP $VENDOR_A_IP $VENDOR_B_IP
8	If			! $success
9	Trace			Vendor @ $DUT_IP did not meet traffic validation criteria
10	Trace			Failed
11	Return			0
12	EndIf			
13	Trace			Vendor @ $DUT_IP meets traffic validation criteria
14	Trace			Passed
15	Return			1
16	Procedure		Numeric	VendorSelectionSetup {String:DUT_IP String:VENDOR_A_IP String:VEND
30	Procedure		Numeric	VendorASetup {String:VENDOR_A_IP String:VENDOR_A_IF}
47	Procedure		Numeric	VendorBSetup {String:VENDOR_B_IP String:VENDOR_B_IF}
64	Procedure		Numeric	VendorCSetup {String:VENDOR_C_IP String:VENDOR_C_IF}
81	Procedure		Numeric	VendorSelectionXmitAndVerify {String:DUT_IP String:VENDOR_A_IP String:
82	If			$DUT_IP eq $VENDOR_A_IP
83	Assign		C1	1
84	Assign		P1	1
85	Assign		C2	1
86	Assign		P2	2
87	Else			Match If on Step '82'
88	If			$DUT_IP eq $VENDOR_B_IP
101	EndIf			
102	StartSes...	XM2		IxExplorer, Session Start chassis=10.200.134.201 ports="($C1.$P1 $C2.$P
103	Execute	XM2		ScriptGen Apply ports=($C1.$P1) filename=C:/ixexplorer-b2b-scriptgen-port1
104	Execute	XM2		ScriptGen Apply ports=($C2.$P2) filename=C:/ixexplorer-b2b-scriptgen-port2

Figure 62. Traffic Generator Procedure port selection based upon device input arguments

12. Encapsulate the Ixia traffic generator calls into a new procedure **VendorSelectionXmitAndVerify**. The goal of this method is to execute the main body of the vendor selection process and then, based upon the criteria defined for expected behavior, declare the test a success or a failure. By creating a module procedure for the traffic transmission and statistics collection process, different vendor selection criteria tests, such as RFC 2544 style throughput, frame loss, and latency measurements can be substituted in place of the default criteria.

13. Complete the end-to-end vendor verification process by using the **Composer** to construct DUT cleanup procedures to return the vendor's device back to the default "golden" state prior to the execution of the DUT setup procedure. This ensures that the vendor device is returned to a ready state when the **Composer** script is executed in an unattended fashion through the **Test Conductor Scheduler**. The script may be used this way as part of a regression series to obtain multiple data points for inclusion in the final results analysis.

14. Cycle through the vendor selection process for each of the vendors. This can be done by changing the values of the **Composer** script **Test Parameters** or by extending your verification with additional test control flow logic to cycle through the participating vendors automatically for a sequencing solution that does not require any human intervention.



	Command Type	Ses...	Return ...	Command String
1	RunProcedure		isSetup	VendorSelectionSetup $DUT_IP $VENDOR_A_IP $VENDOR_B_IP $VEND
2	If			! $isSetup
7	RunProcedure		success	VendorSelectionXmitAndVerify $DUT_IP $VENDOR_A_IP $VENDOR_B_IP
8	RunProcedure		isCleanup	VendorSelectionCleanup $DUT_IP $VENDOR_A_IP $VENDOR_B_IP $VEN
9	If			! $isCleanup
13	If			! $success
14	Trace			Vendor @ $DUT_IP did not meet traffic validation criteria
15	Trace			Failed
16	Return			0
17	EndIf			
18	Trace			Vendor @ $DUT_IP meets traffic validation criteria
19	Trace			Passed
20	Return			1
21	Procedure		Numeric	VendorSelectionSetup {String:DUT_IP String:VENDOR_A_IP String:VENDO
35	Procedure		Numeric	VendorASetup {String:VENDOR_A_IP String:VENDOR_A_IF}
52	Procedure		Numeric	VendorBSetup {String:VENDOR_B_IP String:VENDOR_B_IF}
69	Procedure		Numeric	VendorCSetup {String:VENDOR_C_IP String:VENDOR_C_IF}
86	Procedure		Numeric	VendorSelectionXmitAndVerify {String:DUT_IP String:VENDOR_A_IP String:
131	Procedure		Numeric	VendorSelectionCleanup {String:DUT_IP String:VENDOR_A_IP String:VENI
145	Procedure		Numeric	VendorACleanup {String:VENDOR_A_IP String:VENDOR_A_IF}
162	Procedure		Numeric	VendorBCleanup {String:VENDOR_B_IP String:VENDOR_B_IF}
179	Procedure		Numeric	VendorCCleanup {String:VENDOR_C_IP String:VENDOR_C_IF}

Figure 63. Vendor Selection script pass/fail criteria evaluates to a single high level success or failure

Test Variables

The test used for illustration purposes in the outlined automation methodology can be scaled up in terms of the number of traffic ports used. The port configurations can be replaced with alternate Ixia traffic generator configurations to match the specific traffic patterns required by the organization conducting a vendor selection backoff.

Test Tool Variables

Table 1: Ixia traffic generator variables

Composer Variable	Description
CHASSIS_IP	Ixia chassis management IP address
C1.P1	Ixia traffic generator *Card.Port ID* for first test traffic interface
C2.P2	Ixia traffic generator *Card.Port ID* for second test traffic interface
P1.Tcl	Ixia traffic generator port configuration for first test traffic interface
P2.Tcl	Ixia traffic generator port configuration for second test traffic interface

BIT_RATE_P1	Input parameter for first device vendor test interface
BIT_RATE_P2	Input parameter for second device vendor test interface

DUT Test Variables

Table 2: Device under test variables

Composer Variable	Description
VENDOR_A_IP	Input parameter for first device vendor management IP address
VENDOR_B_IP	Input parameter for second device vendor management IP address
VENDOR_C_IP	Input parameter for third device vendor management IP address
DUT_IP	IP address selector for choosing between vendor device procedures
VENDOR_A_IF	Input parameter for first device vendor test interface
VENDOR_B_IF	Input parameter for second device vendor test interface
VENDOR_C_IF	Input parameter for third device vendor test interface

Results Analysis

Vendor selection process metrics within a **Composer** script are available for export into **Test Conductor** test trend reports. Test trend reports can be used to generate charts of metric values versus time. Figure 64 is an example of throughput versus test run start time as measured over several execution runs of the **Composer** script.

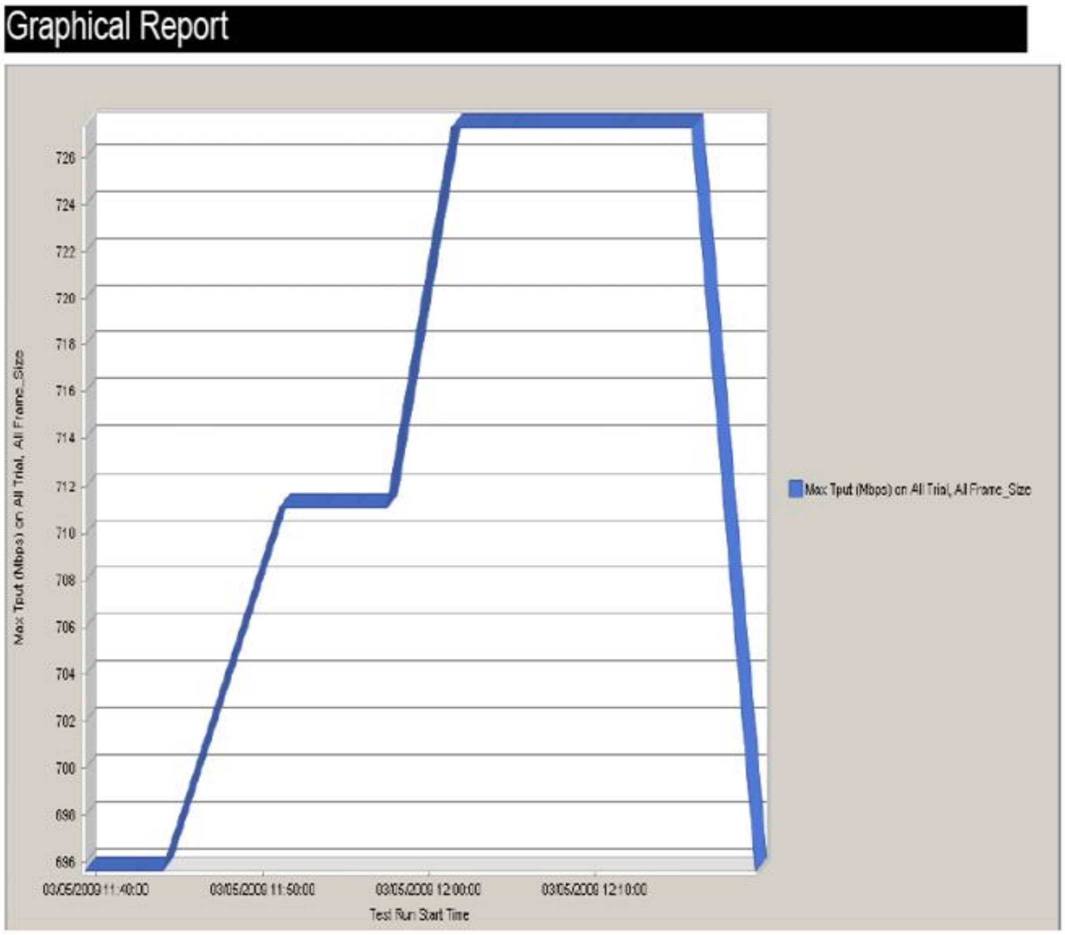


Figure 64. Test Conductor Test Trend Report charting vendor throughput values against time

A vendor selection process often consists of many **Composer** scripts executed as part of a **Test Conductor** regression. A regression trend report charts the overall health of a vendor device as it compares the number of attempted test cases and the number of successfully passed tests cases versus time.

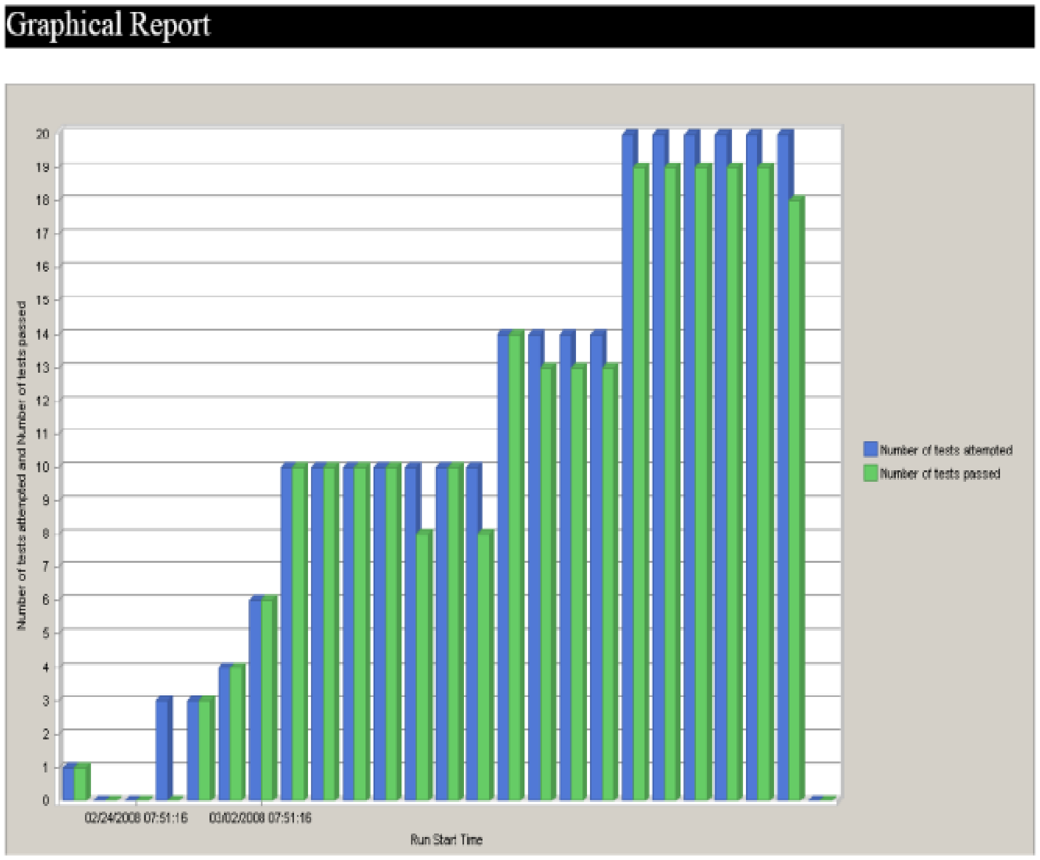


Figure 65. Test Conductor regression trend report charting tests attempted and passed against time

Troubleshooting and Diagnostics

Table 3: Troubleshooting Tips

Issue	Category	Troubleshooting Solution
Variable $var undefined on playback	Composer Editor	Check that syntax of referenced variable is valid. Check that variable exists in current scope. Check that variable initialization has been highlighted and played at least once prior to use.
Variable $var not updated on playback	Composer Editor	Click Reset TCL Interpreter button to clear variable state Replay step that initializes the variable again to update to current value.
Device command response has changed since first captured	Composer Editor	Click Command Response Tab for the step whose response you want to clear. Click the remove button in the upper right of tab. Replay the step to capture the current device response
Variables not listed in Trend Report template	Composer Editor	Check that variable listed in Composer Exported Stats window and that it has been checked.
Error executing command	Composer Debugger	Set a breakpoint on failed execution step. Use Expressions tab to print out, modify and test simple expressions without leaving debugger

Conclusions

Test Composer provides an integrated development environment (IDE) for quality assurance and other IP network test professionals. It is a graphic tool that constructs dynamic tests with logic and flow control. This makes the process easier to design and implement.

Applying **Test Composer** scripts to the automated execution of a vendor selection process yields a more repeatable and accurate set of results that can be used to make a fair, empirically-based vendor selection.

Test Case: Leveraging SNMP Verification in an End-to-End Automation Process

Overview

SNMP (Simple Network Management Protocol) is a structured data protocol that devices use for configuration and transmitting alerts. SNMP integration into a testing methodology is need in many facets of the end-to-end automation process. SNMP is used in the DUT setup phase to set key configuration parameters with default values required by a testing phase. It is also used to retrieve the DUT's operational status in order to check that the device is ready to proceed with the next phase of testing.

SNMP **Get** commands may be used throughout the test algorithm phase to monitor dynamic DUT values to ensure that they are within acceptable tolerances. These include interface metrics, CPU utilization, etc. In addition, SNMP includes **Traps** and **Informs** that may be generated at any time. These notifications indicate normal operational events or more serious error conditions. These conditions could either prevent successful test completion or otherwise invalidate the results of a test.

In some test scenarios, the SNMP implementation itself may be the object of testing, rather than being used as an agent for testing. As part of the DUT cleanup phase, the device is returned to a known good state using **Set** commands to apply steady state values.

Objective

For this example, **Composer** will be used to configure SNMP access to a DUT and also to generate traps as part of a traffic pattern test. Statistics will be collected from the DUT via SNMP **Get** and SNMP **Traps.** These will be used in conditional logic flow statements to determine if a particular throughput value measured on the test interfaces is acceptable. Failure may be declared if the metrics collected through SNMP are found to be outside of acceptable tolerances and/or inconsistent with metrics empirically observed by the traffic generator.

Setup

The **Composer** script generated will be organized into high-level methods that correspond to best practices for SNMP DUT configuration and verification of asynchronous trap events. This graphical script will then be used by the **Test Conductor Regression Manager** and **Scheduler** to run the SNMP verification process multiple times in order to determine a clear, repeatable pattern in the SNMP DUT verification process.

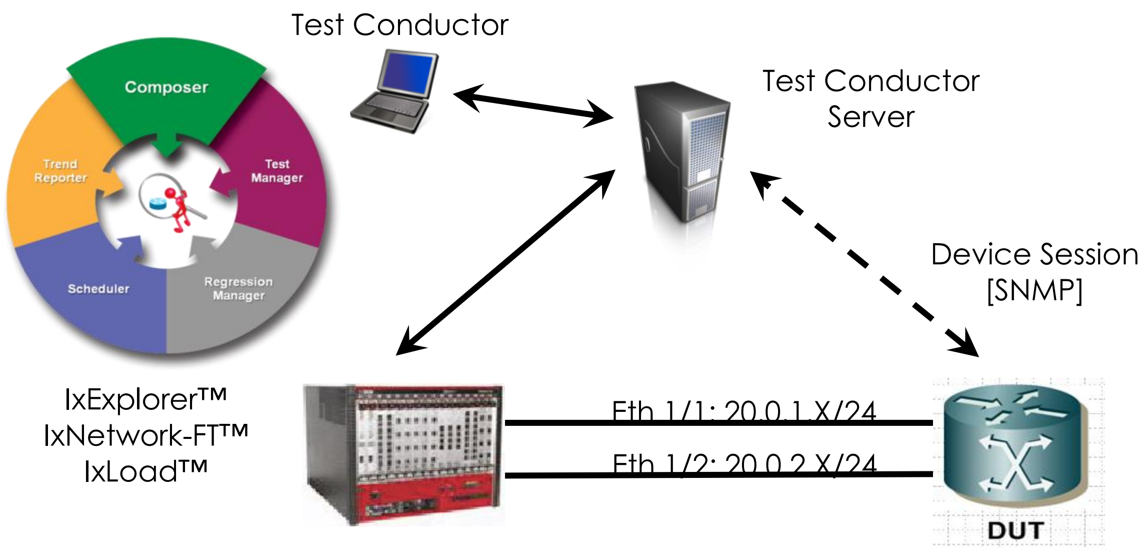


Figure 66. Composer Script SNMP Session Logical Diagram

Step by Step instructions

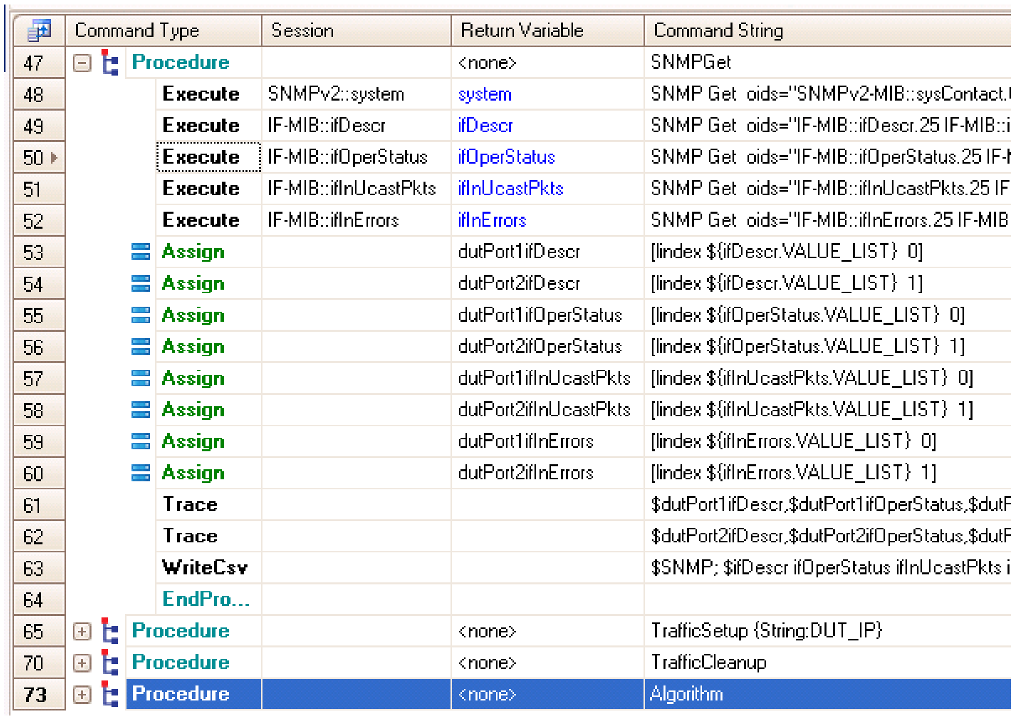
1. Use the **Composer Session Manager** to log in to the DUT. Enable test traffic interfaces, clear statistics counters, and enable the SNMP daemon. Configure the community strings that external hosts will use to access the DUT. Specify access permissions for external hosts, including read-only or read/write access to those communities. Specify SNMP protocol version and optional version 3 authentication information. These are common configuration options across all SNMP devices.

2. Use the **Composer Editor** to encapsulate the DUT initialization process into a **DUT-Setup** procedure that can be a used in other **Composer** scripts that require SNMP access to the DUT. Identify portions of the configuration that can be extracted out as variables. These variables typically correspond to some or all of the options used to initialize SNMP on the DUT. The community string, access permissions, and version information are all potential variables that can be passed into the **DUT-Setup** procedure.

3. Use the **SNMP Resource Manager** to obtain a graphical view of available MIB (Management Information Base) object trees that can be accessed via SNMP. Select MIBs from the existing list or import custom enterprise MIBs. Use the **Composer Session Manager** to establish an SNMP session with the DUT using the same protocol configuration options that were applied on the DUT, such as SNMP community string, protocol version, and access permissions.

4. Use the **Composer Editor** to encapsulate the SNMP session initialization process into an SNMP-Setup procedure. This procedure will need to include input arguments for all of the variable properties previously mentioned, so that this high-level method may be executed against other devices in the same device family.

5. Use the **Composer Editor** to create an **SNMP-Get** procedure to retrieve OID (Object Identifier) values from the DUT. **Composer** assists the script writer in the process of specifying the list of OID values to retrieve by displaying a graphical view of the MIB sources selected in the resource manager. The MIB source trees can be browsed to select the OID values desired. Specify return variables to store the values returned by the DUT. SNMP **Get** commands behave just like other statistics/values collection mechanisms in the **Composer**. The return variables that are populated by the call to SNMP **Get** can be used in control flow logic statements as well as in conditional statements to test against expected values for a result of success or failure.



	Command Type	Session	Return Variable	Command String
47	Procedure		<none>	SNMPGet
48	Execute	SNMPv2::system	system	SNMP Get oids="SNMPv2-MIB::sysContact.(
49	Execute	IF-MIB::ifDescr	ifDescr	SNMP Get oids="IF-MIB::ifDescr.25 IF-MIB::i
50	Execute	IF-MIB::ifOperStatus	ifOperStatus	SNMP Get oids="IF-MIB::ifOperStatus.25 IF-I
51	Execute	IF-MIB::ifInUcastPkts	ifInUcastPkts	SNMP Get oids="IF-MIB::ifInUcastPkts.25 IF
52	Execute	IF-MIB::ifInErrors	ifInErrors	SNMP Get oids="IF-MIB::ifInErrors.25 IF-MIB
53	Assign		dutPort1ifDescr	[lindex ${ifDescr.VALUE_LIST} 0]
54	Assign		dutPort2ifDescr	[lindex ${ifDescr.VALUE_LIST} 1]
55	Assign		dutPort1ifOperStatus	[lindex ${ifOperStatus.VALUE_LIST} 0]
56	Assign		dutPort2ifOperStatus	[lindex ${ifOperStatus.VALUE_LIST} 1]
57	Assign		dutPort1ifInUcastPkts	[lindex ${ifInUcastPkts.VALUE_LIST} 0]
58	Assign		dutPort2ifInUcastPkts	[lindex ${ifInUcastPkts.VALUE_LIST} 1]
59	Assign		dutPort1ifInErrors	[lindex ${ifInErrors.VALUE_LIST} 0]
60	Assign		dutPort2ifInErrors	[lindex ${ifInErrors.VALUE_LIST} 1]
61	Trace			$dutPort1ifDescr,$dutPort1ifOperStatus,$dutF
62	Trace			$dutPort2ifDescr,$dutPort2ifOperStatus,$dutF
63	WriteCsv			$SNMP; $ifDescr ifOperStatus ifInUcastPkts i
64	EndPro...			
65	Procedure		<none>	TrafficSetup {String:DUT_IP}
70	Procedure		<none>	TrafficCleanup
73	Procedure		<none>	Algorithm

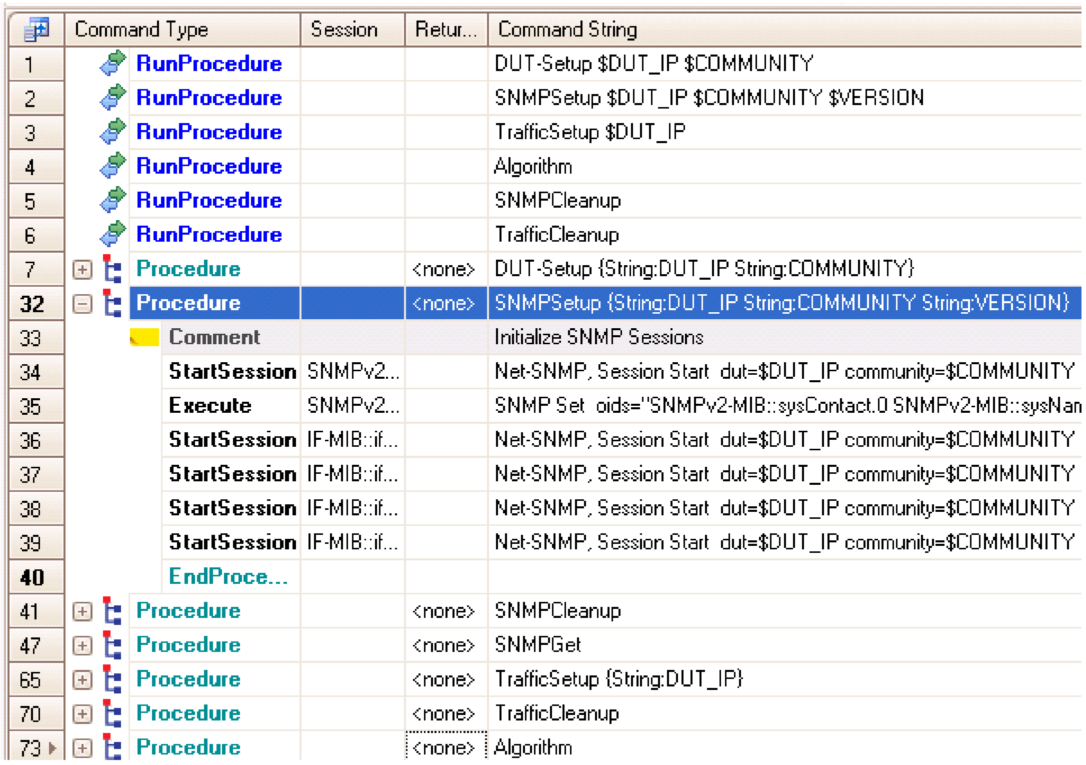
Figure 67. Composer SNMP Get procedure for collecting and logging OID values on DUT

6. Use the **Composer Editor** to create an **SNMP-Set** procedure to change OID values on the DUT. Again, the **Composer** assists the script writer in the process of specifying the list of

OIDs to change. Use the MIB source tree browser to select desired OIDs and obtain meta-data associated with the OIDs such as data type information.

7. Use the **Composer Editor** to create an **SNMP-Trap** procedure to catch both informational and exceptional events that occur on the DUT, which can be collected while the device is under load. This procedure will register specific SNMP OIDs to monitor. The events generated for these OIDs will be collected into a trap queue for analysis by **Composer**. Within this procedure, trap events are matched with corresponding actions depending on the severity level of the trap event. Typical actions in response to a trap message include ignoring the event, calling a procedure to process the event, or even terminating the entire Composer script.

8. Use the **Composer Debugger** to actively monitor SNMP **Trap** events that are received from a single or multiple sources. **Composer** provides simultaneous access to both individual session output logs and aggregated output from all sessions. These are gathered into a single view so you can conveniently correlate what is happening on different devices as they generate SNMP events, comparing them to where the Ixia traffic generator is in the current traffic pattern sequence.



	Command Type	Session	Retur...	Command String
1	RunProcedure			DUT-Setup $DUT_IP $COMMUNITY
2	RunProcedure			SNMPSetup $DUT_IP $COMMUNITY $VERSION
3	RunProcedure			TrafficSetup $DUT_IP
4	RunProcedure			Algorithm
5	RunProcedure			SNMPCleanup
6	RunProcedure			TrafficCleanup
7	Procedure		<none>	DUT-Setup {String:DUT_IP String:COMMUNITY}
32	Procedure		<none>	SNMPSetup {String:DUT_IP String:COMMUNITY String:VERSION}
33	Comment			Initialize SNMP Sessions
34	StartSession	SNMPv2...		Net-SNMP, Session Start dut=$DUT_IP community=$COMMUNITY
35	Execute	SNMPv2...		SNMP Set oids="SNMPv2-MIB::sysContact.0 SNMPv2-MIB::sysNan
36	StartSession	IF-MIB::if...		Net-SNMP, Session Start dut=$DUT_IP community=$COMMUNITY
37	StartSession	IF-MIB::if...		Net-SNMP, Session Start dut=$DUT_IP community=$COMMUNITY
38	StartSession	IF-MIB::if...		Net-SNMP, Session Start dut=$DUT_IP community=$COMMUNITY
39	StartSession	IF-MIB::if...		Net-SNMP, Session Start dut=$DUT_IP community=$COMMUNITY
40	EndProce...			
41	Procedure		<none>	SNMPCleanup
47	Procedure		<none>	SNMPGet
65	Procedure		<none>	TrafficSetup {String:DUT_IP}
70	Procedure		<none>	TrafficCleanup
73	Procedure		<none>	Algorithm

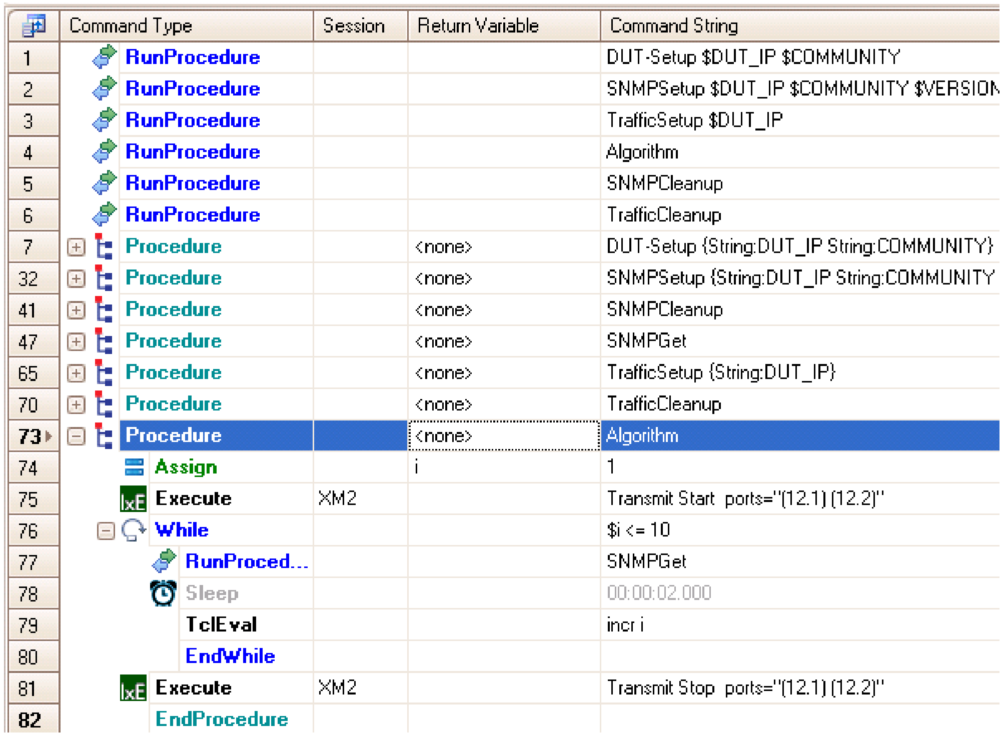
Figure 68. Composer SNMP Setup procedure for initializing sessions with community strings, versions, etc.

9. Use the **Composer** to capture the Ixia traffic generator commands needed to inject the traffic pattern into the DUT. This corresponds to the main body of your test. **Composer**

makes it possible to take a saved traffic generator port configuration and apply it to a different set of ports. Just as was done for the DUT setup, change appropriate hard-coded values such as chassis card and port to parameters that can be selected to match with the current vendor under test, using a selector such as DUT IP Address.

10. Encapsulate the Ixia traffic generator initialization calls into a new procedure named **TrafficSetup**. The goal of this method is to load the traffic generator with the appropriate port configuration onto ports, make them active, and reset metrics and counters prior to the execution of the main test body.

11. Encapsulate the Ixia traffic generator execution calls into a new procedure named **Algorithm**. This is the core of the traffic generation loop. By creating a module procedure for the traffic transmission and statistics collection process, different vendor selection criteria tests such as RFC 2544 style throughput, frame loss, and latency measurements can be substituted in place of the default criteria. Calls to an SNMP **Get** procedure to retrieve specific OID values are inserted at appropriate points in the algorithm. These values are used for comparison against expected values that assist the decision to continue or stop executing traffic, as well as for populating variables that will be used to generate output.



	Command Type	Session	Return Variable	Command String
1	RunProcedure			DUT-Setup $DUT_IP $COMMUNITY
2	RunProcedure			SNMPSetup $DUT_IP $COMMUNITY $VERSIO
3	RunProcedure			TrafficSetup $DUT_IP
4	RunProcedure			Algorithm
5	RunProcedure			SNMPCleanup
6	RunProcedure			TrafficCleanup
7	Procedure		<none>	DUT-Setup {String:DUT_IP String:COMMUNITY}
32	Procedure		<none>	SNMPSetup {String:DUT_IP String:COMMUNITY
41	Procedure		<none>	SNMPCleanup
47	Procedure		<none>	SNMPGet
65	Procedure		<none>	TrafficSetup {String:DUT_IP}
70	Procedure		<none>	TrafficCleanup
73	Procedure		<none>	Algorithm
74	Assign		i	1
75	Execute	XM2		Transmit Start ports="{1 2 1} {1 2 2}"
76	While			$i <= 10
77	RunProced...			SNMPGet
78	Sleep			00:00:02.000
79	TclEval			incr i
80	EndWhile			
81	Execute	XM2		Transmit Stop ports="{1 2 1} {1 2 2}"
82	EndProcedure			

Figure 69. Composer script traffic algorithm body with periodic calls to SNMP procedure

12. Calls to the **SNMP-Trap** procedure are also executed in parallel with the Ixia traffic generator traffic and capture events, since they occur asynchronously. The procedure also assists in determining if the traffic loop should continue executing or if an exceptional event has occurred. This may be a condition that cannot be ignored, requiring that the traffic must terminate, and the test must be declared a failure.

13. Use the **Composer** to encapsulate the session resets and commands needed to return each session into a **Cleanup** procedure. There should be **Cleanup** procedures for each device session that resets the DUT, the Ixia traffic generator and any SNMP sessions that are no longer being used at the end of the script.

Test Variables

The test used for illustration purposes in the outlined automation methodology can be scaled up to increase the number of traffic ports used. The port configurations can be replaced with alternate Ixia traffic generator configurations to match the specific traffic patterns required by the organization conducting a vendor selection backoff. The choice of SNMP OIDs to use will vary depending on which MIBs are used and which metrics are relevant to the traffic being executed.

Test Tool Variables

Table 2: Ixia Traffic Generator Variables

Composer Variable	Description
CHASSIS_IP	Ixia traffic generator management IP address
C1.P1	Ixia traffic generator *Card.Port ID* for first test traffic interface
C2.P2	Ixia traffic generator *Card.Port ID* for second test traffic interface
P1.Tcl	Ixia traffic generator port configuration for first test traffic interface
P2.Tcl	Ixia traffic generator port configuration for second test traffic interface
FRAMES_SENT_P1	Ixia traffic generator total packets transmitted on first traffic interface
FRAMES_SENT_P2	Ixia traffic generator total packets transmitted on second traffic interface

DUT Test Variables

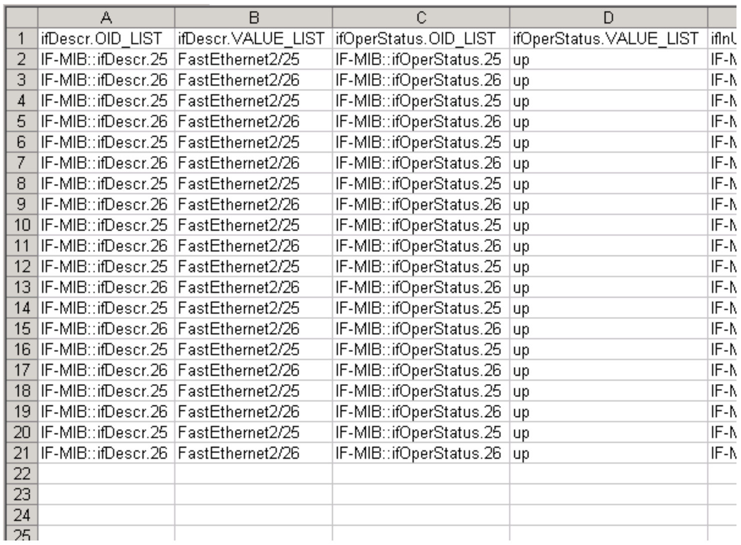
Table 2: Device under test variables

Composer Variable	Description
DUT_IP	Device under test management IP address
COMMUNITY	SNMP community string
VERSION	SNMP protocol version
MIB_LIST	MIB sources governing SNMP session
OID_LIST	SNMP OID used as arguments to Gets and Sets
TRAP_IP	SNMP trap host IP Address

ifInUcastPkts.OID_LIST	SNMP OID names for total packets received on DUT interfaces
ifInUcastPkts.VALUE_LIST	SNMP count values for total packets received on DUT interfaces

Results Analysis

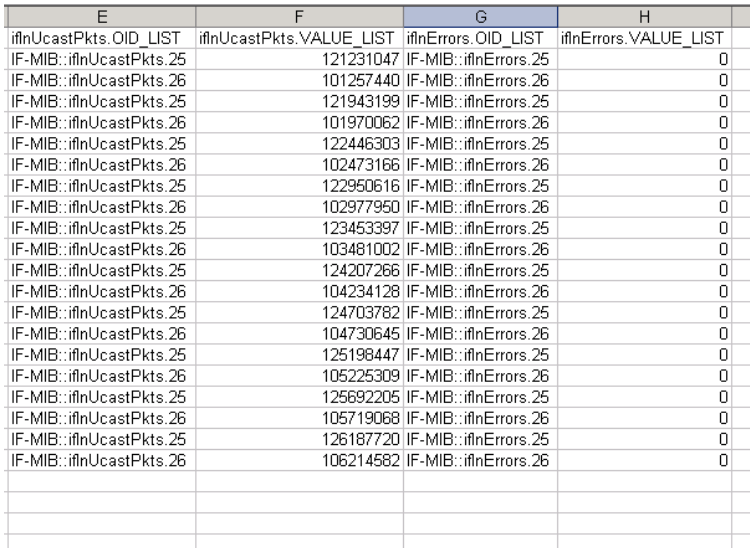
A **Composer** script can be used to generate customer comma separated values (CSV) reports showing the state of a device during the course of the test execution run. SNMP can be used with the standard interface MIB called **IF-MIB**, to check the operational status of the test traffic interfaces on the DUT using the **ifOperStatus OID** and their associated index numbers.



	A	B	C	D	
1	ifDescr.OID_LIST	ifDescr.VALUE_LIST	ifOperStatus.OID_LIST	ifOperStatus.VALUE_LIST	ifInU
2	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
3	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
4	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
5	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
6	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
7	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
8	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
9	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
10	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
11	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
12	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
13	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
14	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
15	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
16	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
17	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
18	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
19	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
20	IF-MIB::ifDescr.25	FastEthernet2/25	IF-MIB::ifOperStatus.25	up	IF-N
21	IF-MIB::ifDescr.26	FastEthernet2/26	IF-MIB::ifOperStatus.26	up	IF-N
22					
23					
24					
25					

Figure 70. Composer CSV with interface operational status values collected via SNMP

SNMP OID values can be collected in a **Composer** session in parallel with another **Composer** session that is generating traffic and collecting Ixia traffic generator statistics. These SNMP OID values can be captured into a CSV file as well as used as values in an expression for a conditional control flow statement such as **If**, **Then**, **Else**, etc. This allows SNMP counter values retrieved from the DUT to be directly correlated with the statistics values observed by the Ixia traffic generator test tool. SNMP can be used to retrieve traffic statistics from the point of view of the DUT through SNMP OIDs such as **ifInUcastPkts** – the number of unicast packets received on an interface. The OID **ifInErrors** provides the number of reception errors that occurred for inbound traffic.



E	F	G	H
ifInUcastPkts.OID_LIST	ifInUcastPkts.VALUE_LIST	ifInErrors.OID_LIST	ifInErrors.VALUE_LIST
IF-MIB::ifInUcastPkts.25	121231047	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	101257440	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	121943199	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	101970062	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	122446303	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	102473166	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	122950616	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	102977950	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	123453397	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	103481002	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	124207266	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	104234128	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	124703782	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	104730645	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	125198447	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	105225309	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	125692205	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	105719068	IF-MIB::ifInErrors.26	0
IF-MIB::ifInUcastPkts.25	126187720	IF-MIB::ifInErrors.25	0
IF-MIB::ifInUcastPkts.26	106214582	IF-MIB::ifInErrors.26	0

Figure 71. Composer CSV with interface counter metrics collected via SNMP

Troubleshooting and Diagnostics

Table 3: Troubleshooting Tips

Issue	Category	Troubleshooting Solution
Variable $var undefined on playback	Composer Editor	Check that syntax of referenced variable is valid. Check that variable exists in current scope. Check that variable initialization has been highlighted and played at least once prior to use.
Variable $var not updated on playback	Composer Editor	Click Reset TCL Interpreter button to clear variable state Replay step that initializes the variable again to update to current value.
Device command response has changed since first captured	Composer Editor	Click Command Response Tab for the step whose response you want to clear. Click the remove button in the upper right of tab. Replay the step to capture the current device response
Variables not listed in Trend Report template	Composer Editor	Check that variable listed in Composer Exported Stats window and that it has been checked.
Error executing command	Composer Debugger	Set a breakpoint on failed execution step. Use Expressions tab to print out, modify and test simple expressions without leaving debugger

Conclusions

Test Composer provides an integrated development environment (IDE) for quality assurance and other IP network test professionals. It is a graphic tool that constructs dynamic tests with logic and flow control. This makes the process easier to design and implement.

Applying **Test Composer** scripts to the automated configuration and event monitoring of SNMP enabled devices greatly increases the confidence that the devices not only achieves the primary throughput, latency, etc. objectives of the traffic plane testing, but that they do so while staying within control plane and operating system configuration tolerance levels. Combining SNMP monitoring with other aspects of a **Composer** script helps to reduce the likelihood that an exceptional event will occur when the device is deployed operationally in the field.

Appendix A: Layer 2-3 Feature Test Automation Cookbook

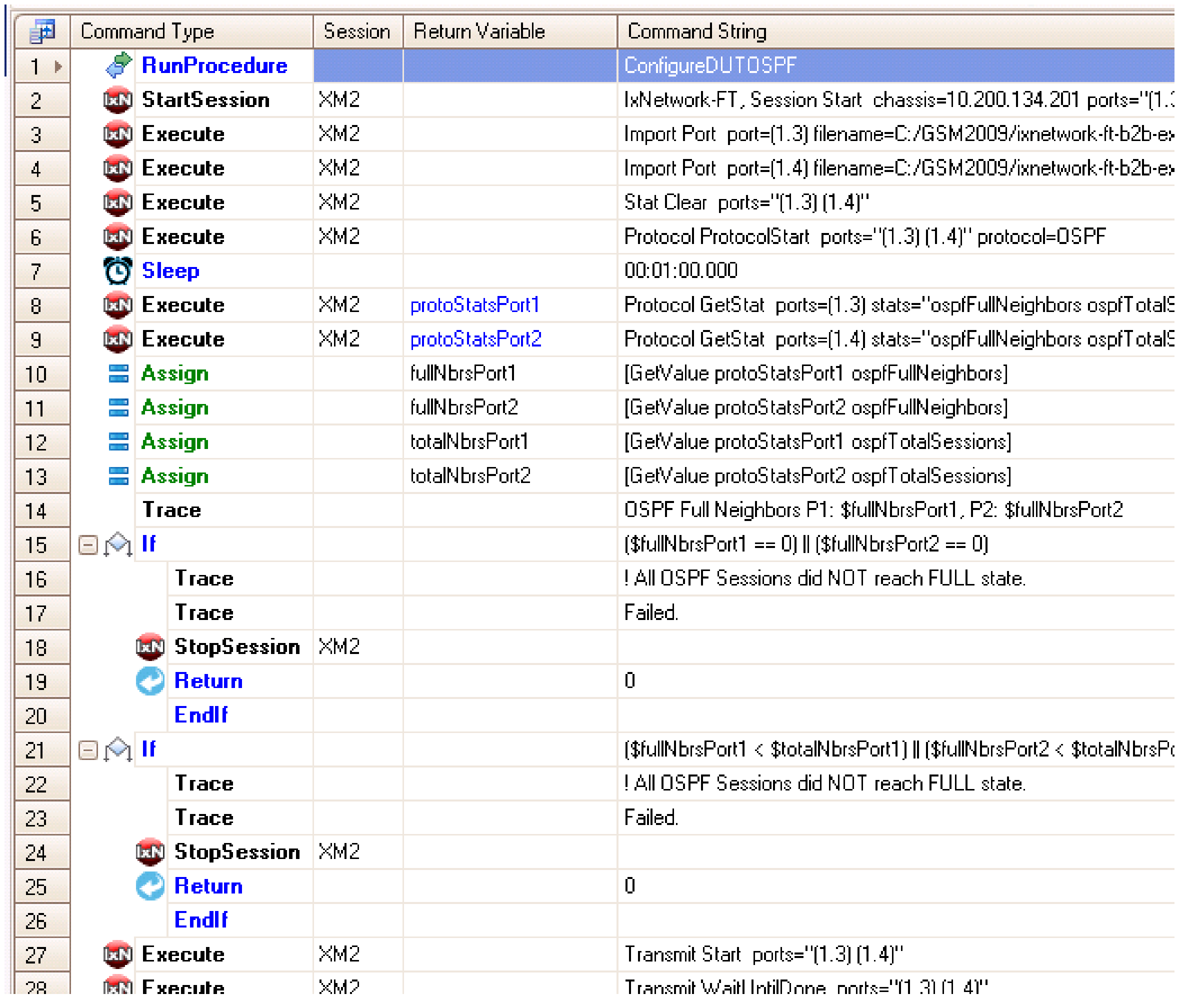
1. Launch the application from the **Test Conductor** Icon. The GUI Console window will be displayed. **Test Composer** is a configuration tool available in the bottom left corner of the screen. Click **Test Composer** to create a new **Composer** test. Click **New Composer Test**. Change the **Name** field value to *IxNetwork-FT-Lab*. Leave the remaining options on this view with their default values. You have now created a new composer test.

2. Click **Steps**, and then click **Show Sessions Console** to display the **Capture** pane. Change the **Name** field value to *XM2*. Select **IxNetwork-FT** from the **Type** list. Enter the IP Address or Hostname for your chassis in the **Chassis** field. Enter the IDs of two ports to which you will apply stream configuration into the **Ports** field using (C.P1 C.P2) notation where C = card #, P1 = port1 #, P2 = port2 #. For example: (1.3 1.4). Change the **Login Name** value to *testconductor*. Browse for the top level IxOS Path for the version of IxOS you are using. Keep in mind that you may be using multi-version IxOS. Example: *C:/Program Files/Ixia/IxOS/5.20.401.68-EB*. You have now entered all the information required before establishing a feature test session.

3. Click **Connect** to establish an active session to the Ixia Chassis Tcl Server. The session will take ownership of the ports you provided and generate a message in the capture window indicating success or failure. Line #1 has been automatically created In the **Test Steps** window with a **Command Type** of **StartSession**. You are now actively connected to a feature test session.

4. Click **Insert Step Below** to create line #2. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *XM2*. Click the **Command String** list to browse for the **Import Port** command. Browse for the first of your chassis ports to populate the **Port** argument. Browse for *C:/ /ixnetwork-ft-b2b-export-port1.prt* to populate the **Filename** argument. The .prt file contains the port configuration for the first port. Click **Check** when done with line #2. Upon completing this step you will have created a script command for loading a port configuration on the first port.

5. Highlight line #2. Click **Copy** and then click **Paste** to create line #3. Click in the **Command String** column to enter text directly. Change the **Port** argument to your second chassis port. Similarly, change the **Filename** argument so that it points to the file *C:/ixnetwork-ft-b2b-export-port2.prt*. Press the **Enter** key to accept your text changes. Click **Save** to save your progress. Upon completing this step you will have created a script command for loading a different port configuration on a second port.



	Command Type	Session	Return Variable	Command String
1	RunProcedure			ConfigureDUTOSPF
2	StartSession	XM2		IxNetwork-FT, Session Start chassis=10.200.134.201 ports="(1.
3	Execute	XM2		Import Port port=(1.3) filename=C:/GSM2009/ixnetwork-ft-b2b-ex
4	Execute	XM2		Import Port port=(1.4) filename=C:/GSM2009/ixnetwork-ft-b2b-ex
5	Execute	XM2		Stat Clear ports="(1.3) (1.4)"
6	Execute	XM2		Protocol ProtocolStart ports="(1.3) (1.4)" protocol=OSPF
7	Sleep			00:01:00.000
8	Execute	XM2	protoStatsPort1	Protocol GetStat ports=(1.3) stats="ospfFullNeighbors ospfTotalS
9	Execute	XM2	protoStatsPort2	Protocol GetStat ports=(1.4) stats="ospfFullNeighbors ospfTotalS
10	Assign		fullNbrsPort1	[GetValue protoStatsPort1 ospfFullNeighbors]
11	Assign		fullNbrsPort2	[GetValue protoStatsPort2 ospfFullNeighbors]
12	Assign		totalNbrsPort1	[GetValue protoStatsPort1 ospfTotalSessions]
13	Assign		totalNbrsPort2	[GetValue protoStatsPort2 ospfTotalSessions]
14	Trace			OSPF Full Neighbors P1: $fullNbrsPort1, P2: $fullNbrsPort2
15	If			($fullNbrsPort1 == 0) \|\| ($fullNbrsPort2 == 0)
16	Trace			! All OSPF Sessions did NOT reach FULL state.
17	Trace			Failed.
18	StopSession	XM2		
19	Return			0
20	EndIf			
21	If			($fullNbrsPort1 < $totalNbrsPort1) \|\| ($fullNbrsPort2 < $totalNbrsPo
22	Trace			! All OSPF Sessions did NOT reach FULL state.
23	Trace			Failed.
24	StopSession	XM2		
25	Return			0
26	EndIf			
27	Execute	XM2		Transmit Start ports="(1.3) (1.4)"
28	Execute	XM2		Transmit WaitUntilDone ports="(1.3) (1.4)"

Figure 72. Composer OSPF Feature Test Script with IxNetwork-FT session commands

6. Click **Insert Step Below** to create line #4. Click the **Command String** list to browse for the **Stat Clear** command. Browse for both your chassis ports to populate the **Port** argument. Click **Check** to accept. Highlight line #2 through line #3 and click **Play** to push the configuration to the ports. Upon completing this step the port configurations will now be active on ports on the Ixia chassis.

7. Highlight line #4 and click **Insert Step Below** to create line #5. Click the **Command String** list to browse for the **Protocol ProtocolStart** command. Browse for both your chassis ports to populate the **Port** argument. Browse for the protocol **OSPF** to populate the **Protocol** argument. Click **Check** to accept. Upon completion of this step you will have created a script command for establishing OSPF adjacencies.

8. Highlight line #5. Click **Insert Step Below** to insert a new step at line #6. Click the **Command Type** list to change the **Command Type** to **Sleep**. Edit the **Minutes** field of the **Command String** to the value of *1* to sleep for 1 minute. Upon completion of this step you will have created a script command for waiting long enough for OSPF adjacencies to have been established to the FULL state.

9. Click **Insert Step Below** to insert a new step at line #7. Click the **Command String** list to browse for the **Protocol GetStat** command. Browse for your first chassis port to populate the **Port** argument. Browse for the **ospfFullNeighbors** and **ospfTotalSessions** statistics to populate the **Stats** argument. Upon completion of this step you will have created a script command for determining the number of active OSPF neighbors and the number of neighbors who successfully reached the FULL state on the first port.

10. Click **Check** when done with arguments for line #7. Enter the value *protoStatsPort1* for the **Return Variable** column. The variable **protoStatsPort1** will be populated with the actual values whenever the step is executed during **Playback** mode. Upon completion of this step you will have specified a container variable to store the OSPF statistics collected for the first port.

11. Click **Copy** and then **Paste** to insert a new step at line #8. Edit the **Command String** text to change the **Port** argument to the second chassis port. Edit the value of the **Return Variable** column to *protoStatsPort2*. The variable **protoStatsPort2** is a convenience variable for storing only the results associated with the second chassis port. Upon completion of this step you will have specified a container variable to store the OSPF statistics collected for a second port.

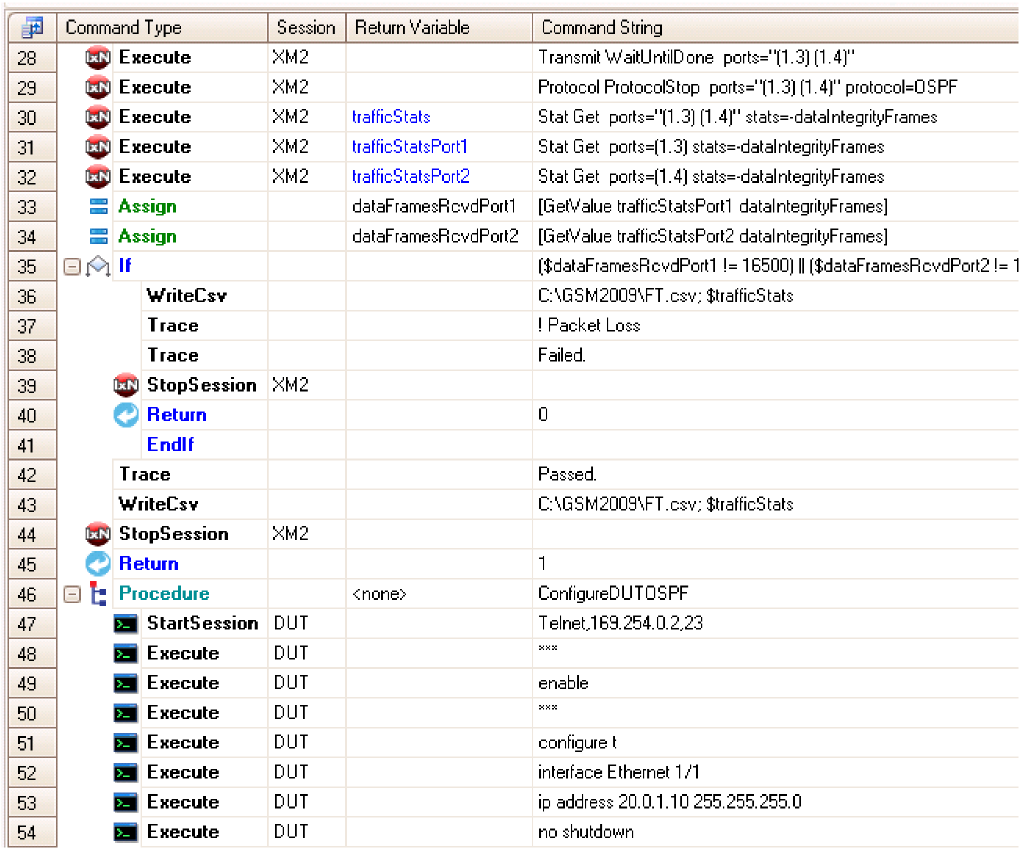
12. Highlight line #7 through line #8. Click **Play** to send the **Protocol GetStat** commands to the Ixia chassis and populate information into the variables you created. As the commands execute, status messages will be appear in the **Execution Errors** window. Remember to click **Save** periodically to save your work. Upon completion of this step you will have collected the current values for OSPF protocol stats and defined the variable structure needed to contain those statistics.

13. Click **Insert Step Below** to create line #9. Click the **Command Type** list to change the **Command Type** to **Assign**. Click the **Command String** list to open the Tcl expression builder. Create the expression *[GetValue protoStatsPort1 ospfFullNeighbors]* by expanding the **protoStatsPort1** tree item under **Test Variables** and double-click the **ospfFullNeighbors** stat and the **Port filter**. Choose the **GetValue** operator from the **Available Commands** list under the **Variable Access** heading in the **Command Category** list. The brackets [ ] can be selected from **Available Operators**. Click **Check** to

accept your changes. Enter *fullNbrsPort1* for the **Return Variable**. Upon completion of this step you will have created a variable containing the number of neighbors on the first port that have reached the FULL state.

14. Click **Copy** and then **Paste** to insert a new step at line #10. Edit the **Command String** text to change the expression to *[GetValue protoStatsPort1 ospfFullNeighbors]* for the second chassis port. Edit the value of the **Return Variable** column to *fullNbrsPort2*. Upon completion of this step you will have created a variable containing the number of neighbors on the second port that have reached the FULL state.

15. Highlight line #9 and line #10. Click **Copy**, then highlight line #10, and use **Paste** to create line #11 and line #12. Edit **Command String** for line #11 to change its value to *[GetValue protoStatsPort1 ospfTotalSessions]*. Edit the **Return Variable** for line #11 to change its value to *totalNbrsPort1*. Upon completion of this step you will have created a variable to contain the number of OSPF neighbors on the first port.



	Command Type	Session	Return Variable	Command String
28	Execute	XM2		Transmit WaitUntilDone ports="{1.3} {1.4}"
29	Execute	XM2		Protocol ProtocolStop ports="{1.3} {1.4}" protocol=OSPF
30	Execute	XM2	trafficStats	Stat Get ports="{1.3} {1.4}" stats=dataIntegrityFrames
31	Execute	XM2	trafficStatsPort1	Stat Get ports={1.3} stats=dataIntegrityFrames
32	Execute	XM2	trafficStatsPort2	Stat Get ports={1.4} stats=dataIntegrityFrames
33	Assign		dataFramesRcvdPort1	[GetValue trafficStatsPort1 dataIntegrityFrames]
34	Assign		dataFramesRcvdPort2	[GetValue trafficStatsPort2 dataIntegrityFrames]
35	If			($dataFramesRcvdPort1 != 16500) \|\| ($dataFramesRcvdPort2 != 1
36	WriteCsv			C:\GSM2009\FT.csv; $trafficStats
37	Trace			! Packet Loss
38	Trace			Failed.
39	StopSession	XM2		
40	Return			0
41	EndIf			
42	Trace			Passed.
43	WriteCsv			C:\GSM2009\FT.csv; $trafficStats
44	StopSession	XM2		
45	Return			1
46	Procedure		<none>	ConfigureDUTOSPF
47	StartSession	DUT		Telnet,169.254.0.2,23
48	Execute	DUT		xxx
49	Execute	DUT		enable
50	Execute	DUT		xxx
51	Execute	DUT		configure t
52	Execute	DUT		interface Ethernet 1/1
53	Execute	DUT		ip address 20.0.1.10 255.255.255.0
54	Execute	DUT		no shutdown

Figure 73. Composer OSPF Feature Test Script with IxNetwork-FT and DUT session commands

16. Edit **Command String** for line #12 to change its value to *[GetValue protoStatsPort2 ospfTotalSessions]*. Edit the **Return Variable** for line #12 to change its value to *totalNbrsPort2*. Upon completion of this step you will have created a variable to contain the number of OSPF neighbors on the second port.

17. Click **Insert Step Below** to create line #13. Click the **Command Type** list to change the **Command Type** to **Trace**. Click the **Command String** list to open the Tcl expression builder. Create the expression *OSPF Full Neighbors P1: $fullNbrsPort1, P2: $fullNbrsPort2* by selecting the appropriate items under **Test Variables** and adding the additional text needed for the debugging output message. Click **Check** to accept your changes. Upon completion of this step you will have created a script command for message output of the number of OSPF neighbors that have successfully reached the FULL state.

18. Click **Insert Step Below** to create line #14. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to **XM2**. Click the **Command String** list to browse for the **Transmit Start** command. Browse for both your chassis ports to populate the **Port** argument. Click **Check** to accept your changes. Upon completion of this step you will have created a script command for starting traffic streams.

19. Click **Insert Step Below** to create line #15. Click the **Command String** list to browse for the **Transmit WaitUntilDone** command. Browse for both your chassis ports to populate the **Port** argument. Click **Check** to accept. Upon completion of this step you will have created a script command for waiting until all traffic streams complete, for streams with a finite end.

20. Click **Insert Step Below** to create line #16. Click the **Command String** list to browse for the **Protocol ProtocolStop** command. Browse for both your chassis ports to populate the **Port** argument. Browse for the protocol **OSPF** to populate the **Protocol** argument. Click **Check** to accept. Upon completion of this step you will have created a script command for terminating OSPF adjacencies.

21. Click **Insert Step Below** to insert a new step at line #17. Click the **Command String** list to browse for the **Stat Get** command. Browse for the both of your chassis ports to populate the **Port** argument. Browse for the **dataIntegrityFrames** statistic to populate the **Stat** argument. Upon completion of this step you will have created a command for obtaining traffic metrics.

22. Click **Check** when done with arguments for line #17. Enter the value *trafficStats* for the **Return Variable** column. The variable **trafficStats** will be populated with the actual values from IxExplorer whenever the step is executed during **Playback** mode. Upon completion of this step you will have created a container variable for storing traffic metrics.

23. Click **Insert Step Below** to insert a new step at line #18. Click the **Command String** list to browse for the **Stat Get** command. Browse for only the first of your chassis ports to populate the **Port** argument. Browse for the **dataIntegrityFrames** statistic to populate the

Stat argument. Upon completion of this step you will have created a script command for collecting the traffic metrics on the first port.

24. Click **Check** when done with arguments for line #18. Enter the value *trafficStatsPort1* for the **Return Variable** column. The variable **trafficStatsPort1** is a convenience variable for storing only the results associated with the first chassis port. Upon completion of this step you will have created a container variable for storing traffic metrics from the first port.

25. Click **Copy**, highlight line #18, and then **Paste** to insert a new step at line #19. Edit the **Command String** text to change the **Port** argument to the second chassis port. Edit the value of the **Return Variable** column to *trafficStatsPort2*. The variable **trafficStatsPort2** is a convenience variable for storing only the results associated with the second chassis port. Upon completion of this step you will have created a container variable for storing the traffic metrics for the second port.

26. Highlight line #17 through line #19. Click **Play** to send the **Stat Get** commands to the Ixia chassis and populate information in the variables you created. As the commands execute status messages will appear in the **Execution Errors** window. Remember to click **Save** periodically to save your work. Upon completion of this step you will have collected traffic metrics and filled in the value and variable structures created in previous steps.

27. Click **Insert Step Below** to create line #20. Click the **Command Type** list to change the **Command Type** to **Assign**. Click the **Command String** list to open the Tcl expression builder. Create the expression *GetValue trafficStatsPort1 dataIntegrityFrames* by expanding the **trafficStatsPort1** tree item under Test Variables and double-click the **dataIntegrityFrames** stat. The brackets [ ] may be selected from **Available Operators**. Click **Check** to accept your changes. Enter *dataFramesRcvdPort1* for the **Return Variable**. Upon completion of this step you will have created a container variable for the traffic frames received on the first port.

28. Click **Copy** and then **Paste** to insert a new step at line #21. Edit the **Command String** text to change the expression to *GetValue trafficStatsPort2 dataIntegrityFrames* for the second chassis port. Edit the value of the **Return Variable** column to *dataFramesRcvdPort2*. Click **Insert Step Below** to create line #22. Click the **Command Type** list to change the **Command Type** to **Trace**. Edit the **Command String** text to output *Passed* as a debugging message. Upon completion of this step you will have created a container variable for the traffic frames received on the second port.

29. Click **Insert Step Below** to create line #23. Click the **Command Type** list to change the **Command Type** to **WriteCSV**. Click the **Command String** list to open the argument builder. Specify the expression *C:\FT.csv* for the File Name argument. Browse for the **trafficStats** item to populate the **Variable** argument. Click **Check** to accept your changes. Upon completion of this step you will have created a script command for the output of traffic statistics to a comma separated variable file.

30. Click **Disconnect** to close the IxExplorer session and it will automatically create line #24 with the **Command Type** of **StopSession**. Insert a **Return** statement at line #25 with **Command String** value set to *1*. Make sure to click **Save** to save this version of the IxExplorer script before proceeding. Upon completion of this step you will have disconnected from the feature test session.

31. Click **Debug** to switch to the Debugger. Click the **Line #** column next to line #4, line #14 and line #22 to add a breakpoint that will pause the execution at each of those lines. You may add additional breakpoints to other lines where you are interested in pausing the execution of your script. Upon completion of this step you will have identified the locations in the **Composer** script where execution will be paused upon reaching those points.

32. If you wish to disable a breakpoint, simply click **Breakpoint** and it will remove the fill from the circle, indicating that the breakpoint is present, but disabled. Upon completion of this step you will have identified any locations in the composer script where the execution pause points can be temporarily disabled.

33. Click **Play** to start the execution of the test. The test will execute until it hits the first breakpoint. You can open IxNetwork-FT and refresh the chassis view to see the configuration that has been applied to this point. Upon completion of this step IxExplorer will show the effects of having executed script commands in Composer.

34. Click **Play** to continue execution to the next breakpoint and observe the effect in IxNetwork-FT where appropriate. Make sure that you are able to reach the breakpoint associated with the **Passed** trace message on line #22 before continuing further. If you can successfully reach this breakpoint then your script is working as expected. Click **Play** to execute to the end. You can use the **Global Output** window to assist you in verifying the correct execution of your script. Upon completion of this step you will have successfully executed a feature test scenario in an automated fashion using Composer. The script may be copied and modified to automate the feature test of other protocols.

Appendix B: Layer 4-7 Applications Automation Cookbook

1. Launch the application from the **Test Conductor** icon. The GUI Console window will be displayed. **Test Composer** is a configuration tool available in the bottom left corner of the screen. Upon completion of this step you will have successfully accessed the Test Conductor server via the client.

2. Click **Test Composer** to create a new **Composer** test. Click **New Composer Test**. Change the **Name** field value to *IxLoad-Lab*. Leave the remaining options on this view with their default values. Click **Steps**. Click S**how Sessions Console** to make the **Capture** pane viewable. Change the **Name** field value to *XM2*. Select **IxLoad** from the **Type** list. Upon completion of this step you will have initialized a new composer test.

3. Browse for *C:/ixload-b2b-http-tput.rxf* to populate the **Repository File** argument. Enter the IDs of 2 ports to which you will apply IxLoad configuration into the **Port List** field using Chassis(C.P1) Chassis(C.P2) notation where Chassis = IP Address or Hostname, C = card #, P1 = port1 #, P2 = port2 #. For example: *10.200.134.170(1.5 1.6)*. Upon completion of this step you will have entered all connection information required to load a default IxLoad configuration from a file.

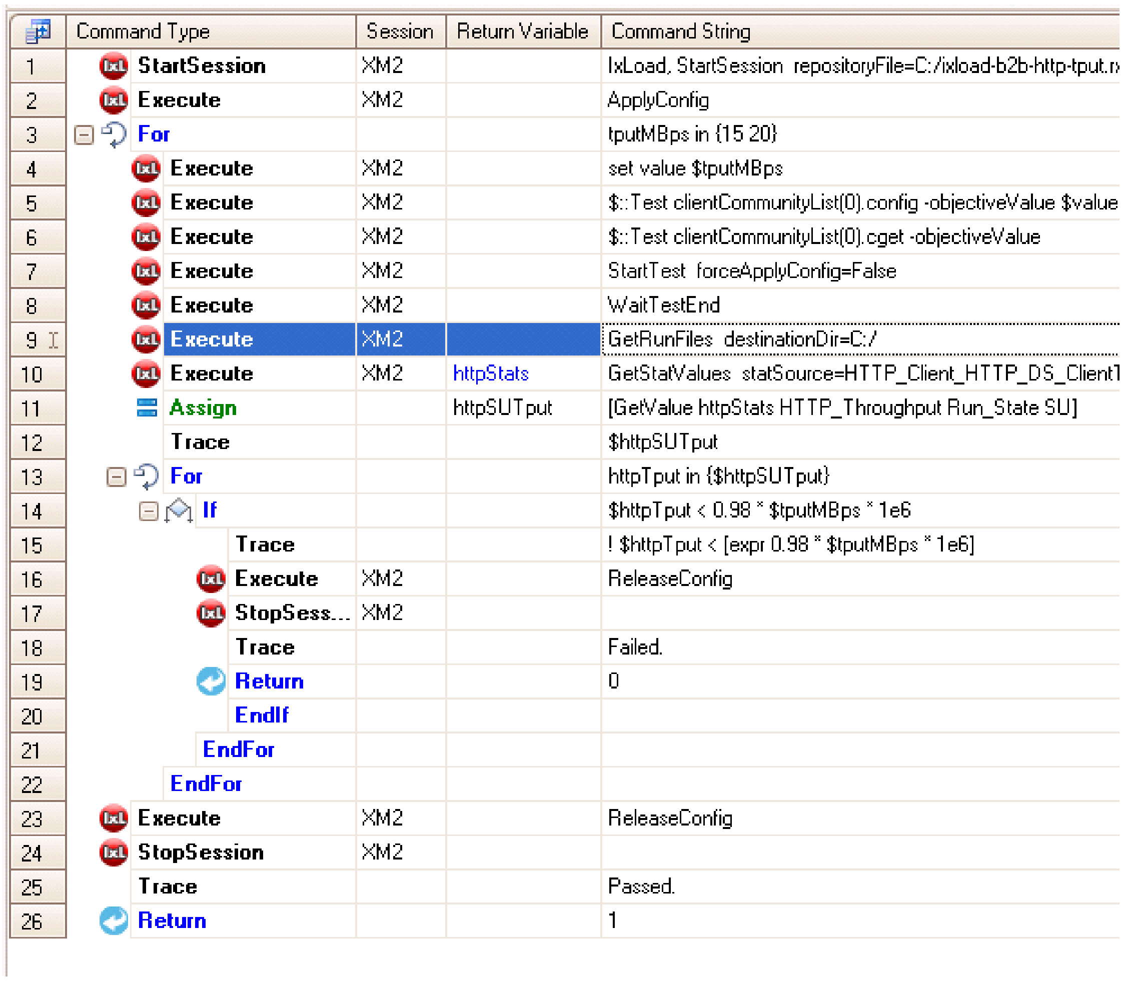
4. Click **Connect** to establish an active session to the Ixia chassis. The IxLoad session will take ownership of the ports you provided and generate a message in the capture window indicating success or failure. Line #1 has been automatically created in the **Test Steps** window with a **Command Type** of **StartSession**. Upon completion of this step you will have established an active application layer session.

5. Click the **Insert Step Below** to create line #2. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *XM2*. Click the **Command String** list to browse for the **ApplyConfig** command. Click **Check** when done with line #2. Upon completion of this step you will have created a script command for applying the loaded configuration to Ixia chassis ports.

6. Click **Insert Step Below** to create line #3. Click the **Command String** list to browse for the **StartTest** command. Click **Check** to accept the default of **False** for the **Force Apply Config** argument. Click **Save** to save your progress. Click **Insert Step Below** to create line #4. Click the **Command String** list to browse for the **WaitTestEnd** command. Click **Check** to accept. Upon completion of this step you will have created a script command for starting an application layer traffic test.

7. Click **Insert Step Below** to create line #5. Click the **Command String** list to browse for the **GetRunFiles** command. Browse for *C:/* to populate the Destination Directory argument. Click **Check** to accept. Upon completion of this step you will have created a command to output the comma separated value files that contain application traffic metric to a user defined location.

8. Click **Insert Step Below** to create line #6. Click the **Command String** list to browse for the **ReleaseConfig** command. Click **Check** to accept. Highlight line #2. Click **Play** to send the **ApplyConfig** command to the Ixia chassis. Upon completion of this step you will have activated the application layer configuration on the Ixia chassis ports.



	Command Type	Session	Return Variable	Command String
1	StartSession	XM2		IxLoad, StartSession repositoryFile=C:/ixload-b2b-http-tput.r
2	Execute	XM2		ApplyConfig
3	For			tputMBps in {15 20}
4	Execute	XM2		set value $tputMBps
5	Execute	XM2		$::Test clientCommunityList(0).config -objectiveValue $value
6	Execute	XM2		$::Test clientCommunityList(0).cget -objectiveValue
7	Execute	XM2		StartTest forceApplyConfig=False
8	Execute	XM2		WaitTestEnd
9	Execute	XM2		GetRunFiles destinationDir=C:/
10	Execute	XM2	httpStats	GetStatValues statSource=HTTP_Client_HTTP_DS_Client1
11	Assign		httpSUTput	[GetValue httpStats HTTP_Throughput Run_State SU]
12	Trace			$httpSUTput
13	For			httpTput in {$httpSUTput}
14	If			$httpTput < 0.98 * $tputMBps * 1e6
15	Trace			! $httpTput < [expr 0.98 * $tputMBps * 1e6]
16	Execute	XM2		ReleaseConfig
17	StopSess...	XM2		
18	Trace			Failed.
19	Return			0
20	EndIf			
21	EndFor			
22	EndFor			
23	Execute	XM2		ReleaseConfig
24	StopSession	XM2		
25	Trace			Passed.
26	Return			1

Figure 74. Composer HTTP Throughput Application Test Script with IxLoad session commands

9. Highlight line #6. Click **Insert Step Above** to insert a new step at line #6. Change the value of the **Command Type** to **Execute**. Change the value of the **Session column** to *XM2*. Click the **Command String** list to browse for the **GetStatValues** command. Upon completion of this step you will have created a command to collect application traffic metrics.

10. Browse for *HTTP_Client_HTTP_DS_ClientTraffic1@ClientNetwork1* to populate the **Stat Source** argument. Browse for *{Run State} {HTTP Throughput}* to populate the **Stat Names** argument. Choose **Custom** from the **Selection** list. Specify * for wildcard in the **Selected Row(s)** argument. Click **Check** to accept. Enter the value *httpStats* for the **Return Variable** column. Upon completion of this step you will have identified the application traffic metrics to be collected.

11. Highlight line #3, line #4 and line #6. Click **Play** to send the **StartTest**, **WaitTestEnd**, and **GetStatValues** commands to the Ixia chassis and populate information in the variable you created. Upon completion of this step you will have executed the application traffic test and collected traffic metrics into a container variable.

12. Highlight line #6. Click the **Insert Step Below** to insert a new step at line #7. Click the **Command Type** list to change the **Command Type** to **Assign**. Click the **Command String** list to open the Tcl expression builder. Upon completion of this step you will have created a script line to create a new variable.

13. Create the expression *[GetValue httpStats HTTP_Throughput Run_State SU]*. You may type the expression directly into the bottom evaluation window if you are familiar with Tcl syntax. If not, you may expand the composite variable listed in the **Test Variables** window and right-click to statistics to apply a filter based on other statistics, as well as choose the [ ] operator from the **Available Operators** list. Click **Check** to accept your changes. Enter the value *httpSUTput* for the **Return Variable** column. Upon completion of this step you will have populated a new variable with the application traffic throughput metrics that correspond to the steady state execution of the application traffic configuration.

14. Click the **Insert Step Below** to insert a new step at line #8. Change the value of the **Command Type** to **Trace**. Click the **Command String** list to open the Tcl expression builder. Create the expression *$httpSUTput* by selecting the appropriate item under **Test Variables**. Upon completion of this step you will have created a script command to output the steady state application throughput.

15. Highlight lines #3 through 8. Click **Place Inside For** to indent these steps inside a new **For** loop beginning at line #3 and ending at line #11. Click the **Command String** list to open the **For** loop expression builder. Enter *tputMBps* for the value of the **Assign** to variable field. Select **SET** from the **Loop** type list. Enter two space separated **SET** values into the **Values** list, *15 20*. Click **Check** to accept your changes. Upon completion of this step you will have constructed a basic iteration over multiple application throughput values.

16. Highlight line #3. Click **Insert Step Below** to insert a new step at line #4. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *XM2*. Enter the expression set value *$tputMBps* directly into the **Command String**. Upon completion of this step you will have copied the iterative application throughput value into a variable used by the application session.

17. Click **Insert Step Below** to insert a new step at line #5. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *XM2*. Click the **Command String** list to clear the **Use Composer Variables** check box. Click **Check** to accept your change. On completion of this step, you will have identified to the application session that the variable should be evaluated within the local context of the application session itself.

18. Enter the expression *$::Test clientCommunityList(0).config –objectiveValue $value* directly into the **Command String**. Click **Insert Step Below** to insert a new step at line #6. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *XM2*. Upon completion of this step you will have created a command to change the current application throughput objective value to the next value in the iteration over throughput values.

19. Click the **Command String** list to clear the **Use Composer Variables** check box. Click **Check** to accept your change. Enter the expression *$::Test clientCommunityList(0).cget –objectiveValue* directly into the **Command String**. Upon completion of this step you will have created a command to retrieve the changed value of the application throughput objective value for verification purposes.

20. Click **Disconnect** to close the IxLoad session; line #15 will be created with a **Command Type** of **StopSession**. Make sure to click **Save** to save this version of the IxLoad script before proceeding. Upon completion of this step you will have terminated the application traffic session.

21. Click **Insert Step Below** to create line #16. Click the **Command Type** list to change the **Command Type** to **Trace**. Edit the **Command String** text to output *Passed* as a debugging message. Insert a **Return** statement at line #17 with the **Command String** value set to *1*. Upon completion of this step you will have identified the exit point for the composer script when execution completes successfully.

22. Highlight line #14 through line #17. Click **Copy**, highlight line #13, and then **Paste** to create line #13 through line #16. Edit **Command String** for line #15 to change its value to *Failed*. Edit **Command String** for line #16 to change its value to *0*. Remember to click **Save** periodically to save your work. Upon completion of this step you will have identified the exit point for the composer script when the execution fails to complete with expected results.

23. Highlight lines #13 through #16. Click **Place Inside If** to indent these steps inside a new line loop beginning at line #13 and ending at line #18. Enter the expression *$httpTput < 0.98 * $tputMBps * 1e6* directly into the **Command String**. Upon completion of this step you will have specified an expression requiring that all steady state throughput needs to be no more than two percent less than the current iterative value being attempted.

24. Click **Insert Step Below** to create line #14. Click the **Command Type** list to change the **Command Type** to **Trace**. Edit the **Command String** text to *output ! $httpTput < [expr 0.98 * $tputMBps * 1e6]* as a debugging message. Upon completion of this step you will

have created an error message indicating that a steady state throughput value was more than two percent less than the current iterative value being attempted.

25. Highlight lines #13 through #19. Click **Place Inside For** to indent these steps inside a new line loop beginning at line #13 and ending at line #21. Build the expression *httpTput* in *{$httpSUTput}* to be evaluated on each iteration of the loop. Upon completion of this step you will have created an iterative loop to collect all of the steady state throughput values that were returned as a list by the stat collection.

26. Click **Debug** to switch to the debugger and set breakpoints at line #18 and line #25. This will allow you to monitor the exit conditions of your test script. Within **Test Composer**, the **Global Output** view enables you to monitor progress for all sessions in one window. Upon completion of this step you will have identified execution pause points to determine which exit point, success or failure, the script will pass through.

27. Click **Play** to start the execution of the test. The test will execute until it hits one of your two breakpoints. Make sure that you are able to reach the breakpoint associated with the *Passed* trace message on line #25. If you can successfully reach this breakpoint, then your script is working as expected. Click **Play** to execute to the end. You can use the **Global Output** window to assist you in verifying the correct execution of your script. Upon completion of this step you will have executed an application traffic test configuration in an automated fashion using **Composer**. The **Composer** script may be copied and modified to load alternate application traffic configuration files and collect alternate stats as required by other protocol traffic types.

Appendix C: IxN2X PPPoE Feature Test Automation Cookbook

This cookbook is provided to show how to approach creating an automated script by using Test Composer and the IxN2X plug-in. The basic methodology can be applied to almost any sort of test. The high-level steps include the following:

1. Configure the IxN2X using the graphical user interface and save the configuration.
2. Using the automation, load the configuration.
3. Start the protocols and wait for them to settle.
4. Use the protocol state statistics to validate that the protocols are up.
5. Configure the traffic stats that you want to collect.
6. Start the traffic and wait for some period of time.
7. Stop the traffic and wait for the packets to complete their transmission.
8. Measure the statistics and validate that the measurements are within the acceptable range.
9. Repeat the steps as necessary with various changes to the configuration as required. For example, variations in frame size and transmission rate are common ways to vary the configuration.

Following are the steps used to perform a basic PPPoE feature test.

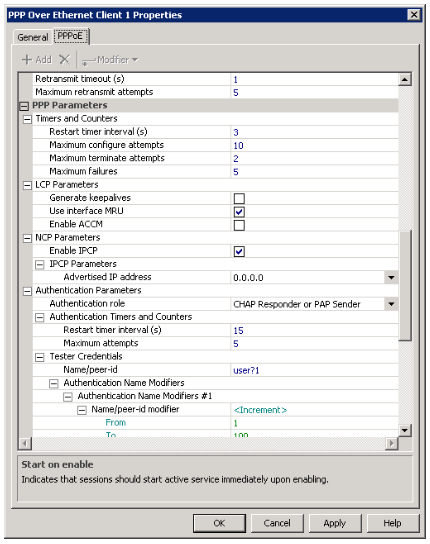
1. Start the application from the **Test Conductor** Icon. The GUI Console window appears. **Test Composer** is a configuration tool available in the bottom left corner of the screen. Click **Test Composer** to create a new **Composer** test. Click **New Composer Test**. Open the **File-Properties** to change the **Name** field value to *IxN2X-Lab*. Leave the remaining options on this view with their default values. Close the **Properties** dialog. You have now created a new composer test.

2. The first stage of building the test is to establish a connection to the controller. This involves configuring and connecting a Test Composer session to the IxN2X Controller.

   a. In the Sessions Tab section of the editor, select **IxN2X** from the **Type** list. Change the **Name** field value to Ix*N2X.* Enter the IP Address or Hostname for your controller in the **IxN2X controller hostname or IP Address** field. Select the version of software that you want to use or leave it as 'Latest' to select the most recent version of IxN2X software installed on the controller. Enter the **IxN2X Test Session Label** to the name that you want to use. The IxN2X session can establish a new session on the controller or it can reconnect to an existing session. Set the **Reuse existing test session** to yes if you want to reconnect to an existing session. If no session exists with the name you entered, one will be created for you. You have now entered all the information required before establishing a feature test session.

   b. Click **Connect** to establish an active session to the IxN2X controller. The session will generate a message in the capture window indicating success or failure. Line #1 has been automatically created In the **Test Steps** window with a **Command Type** of **StartSession**. You are now actively connected to a feature test session.

3. The second phase is to load the test configuration. The configuration of the IxN2X is still done using the IxN2X GUI. In this case, the configuration involves setting up a standard PPPoE configuration using two ports. After you have created and saved the configuration, you are ready to program Test Composer to load that configuration automatically.



Retransmit timeout (s)	1
Maximum retransmit attempts	5
PPP Parameters	
Timers and Counters	
Restart timer interval (s)	3
Maximum configure attempts	10
Maximum terminate attempts	2
Maximum failures	5
LCP Parameters	
Generate keepalives	☐
Use interface MRU	☑
Enable ACCM	☐
NCP Parameters	
Enable IPCP	☑
IPCP Parameters	
Advertised IP address	0.0.0.0
Authentication Parameters	
Authentication role	CHAP Responder or PAP Sender
Authentication Timers and Counters	
Restart timer interval (s)	15
Maximum attempts	5
Tester Credentials	
Name/peer-id	user?1
Authentication Name Modifiers	
Authentication Name Modifiers #1	
Name/peer-id modifier	<Increment>
From	1
To	100

Start on enable
Indicates that sessions should start active service immediately upon enabling.

Figure 75. Configure PPPoE with IxN2X

a. Click **Insert Step Below** to create another step. Change the value of the **Command Type** to **Execute**. Change the value of the **Session** column to *IxN2X*. Click the **Command String** list to browse for the **Session LoadConfigurationFile** command. Browse for the configuration file that you want to load. Enter the **Target ports** field with the ports that you want to use during this test. Use the standard IxN2X syntax such as 101/1 101/2. Or you can leave it blank if the configuration has the ports that you want to use already saved. Click **Ok** when done with the step. On completing this step, you will have created a script command to load a configuration on the controller.

b. Highlight the command that you just created in the previous step and click **Play** to push the configuration to the ports. On completing this step, the port configurations will now be loaded on the ports.

c. For good measure, we recommend you to include a command to send some ARP packets to establish connectivity between the ports. Insert another execute command after the LoadConfigurationFile command and select **Link RunArp** with the **PortList** set to All and the **EnableSelfPaced** set to yes. Highlight the command and click Play.

4. The next phase is to build a command that enables the protocol emulations.

a. Add another Execute command and select **Protocol PppAction.** Next, select the appropriate values from the lists provided by the Command Editor. For the **PortList,** select the port with the PPP Client emulation configured, for example, '101/1,' **Action** set to 'Enable,' **ProtocolEntity** set to 'PPP over Ethernet Client,', and **ItemList** set to 'PPP over Ethernet Client 1.'

b. Next, add a sleep command to allow the protocol to settle before proceeding to the next command. Fifteen seconds is a nominal value, but more or less may be required depending on your configuration.

c. Add another Execute command. This command will refresh the Session IDs. Use the Protocol PppAction command again with the same settings as earlier except this time the **Action** is set to 'Open Connections.'

d. Again, we recommend you to include another sleep command to allow the system to settle.

5. After the sessions have been refreshed, it is good to validate this.

   a. Using the Execute command, add a **GetProtocolStateStatistics** for the port running the PPPoE client, in this example, that would be 101/1. Because the established sessions is the metric to be verified, include **Established** as one of the statistics to query. Make sure to provide a return variable to receive the statistics, such as 'PPP_Stats.'

   b. If multiple statics are queried, the GetValue command is a good way to filter out the other stats so that the specific stat can be verified. Add a **GetValue** command to the test where the stat to be extracted is the **Established** sessions and filtered by the port **101/1**. Again, provide a return variable to receive the value from the GetValue command.

   c. Verify that the number of established sessions match what the test is configured to create. In this test, that value is 100. Using the **If** command, compare the variable that contains the number of established sessions created in the previous step to the number of expected established sessions, in this case, 100. If the two do not match, execute the necessary commands to stop the test and flag the test as failed by returning the value 0 using the **Return** command.

6. If the two values do match, the test can proceed. The next command to execute is to update the mesh stream group.

   a. Use the command **Traffic UpdateMeshStreamGroup** and select the traffic mesh from the list. In this case, it is 'TrafficMesh 1.'

   b. Insert a **Sleep** 15 seconds command.

7. To verify that the protocol is working, traffic needs to be transmitted. To verify that the traffic is being transmitted and received, configure the IxN2X to collect the relevant stats. In this case, the stats of interest are **TEST_PACKETS_TRANSMITTED** and **TEST_PACKETS_RECEIVED** on both ports.

   a. Add another Execute command and use the command string **Statistics ConfigurePortView**. Select for the **portList** both ports, in this case **101/1** and **101/2**. Finally, pick the relevant stats such as **TEST_PACKETS_RECEIVED** and **TEST_PACKETS_TRANSMITTED.** Give the view a name, one that will be used later to get the stats. In this example, the name **Stats** is used.

8. Begin sending traffic.

   a. Using the Execute command, create a **Traffic StartTrafficEngine** with a max wait time big enough to allow the IxN2X to start the traffic engine. Ten seconds is a typical time to wait. If the traffic engine does not begin within that time, an error appears.

   b. Wait for some time for the traffic to be sent. In this example, 30 seconds is sufficient to prove that traffic is successfully sent to the hosts.

   c. Using the Execute command, create a **Traffic StopTrafficEngine** with a max wait time big enough to allow the IxN2X to start the traffic engine. Ten seconds is a typical time to wait. If the traffic engine does not stop within that time, an error appears.

9. Now is the time to collect the traffic statistics that were configured in the earlier step. These stats will be used to determine whether or not the test passed by comparing the transmitted packets to the received packets.

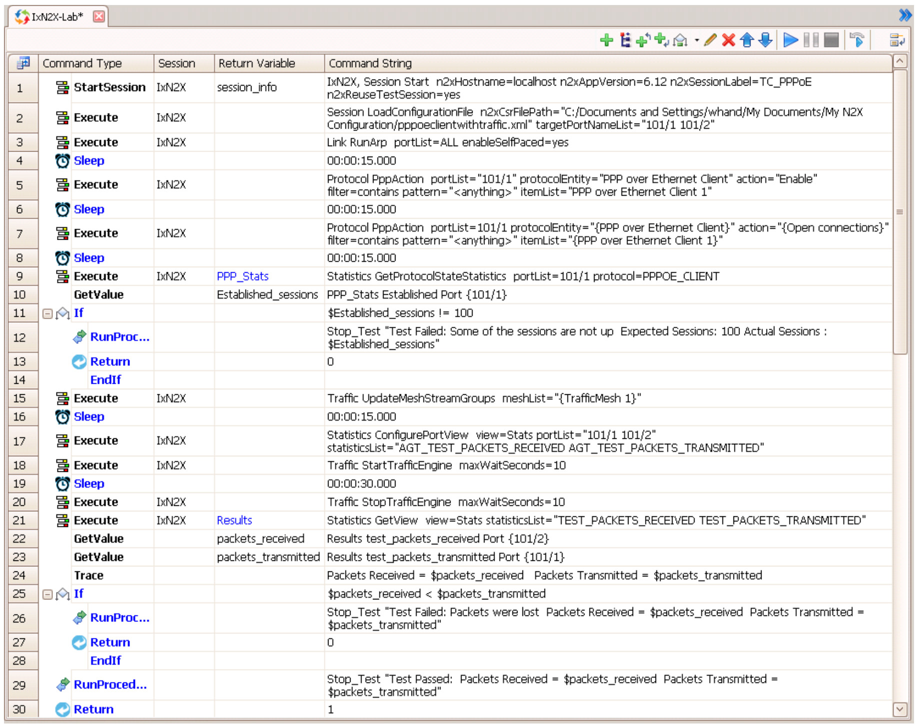
   a. Using the Execute command, create a **Statistics GetView** command with the 'Stats' view name. The view may have more stats that are needed for the test, because additional stats may be useful to have logged with the results of the test, but are not necessary for establishing success or failure. So select just the stats from the view that will be used to measure success. In this case, select the **TEST_PACKETS_RECEIVED** and **TEST_PACKETS_TRANSMITTED** statistics for both ports in the test.

   b. Using the same sort of **GetValue** command as earlier, get the received packets from the receiving port (101/2 in this case) and the transmitted packets from the send port (101/1 in this case).

   c. Compare the two values using the **If** command. If the packets received is less than packets transmitted, fail the test by creating a **Return** command with the value **0.**

   d. If the two values match, use a **Return** command with the value **1** to indicate that the test passed.

10. Throughout the test, it may be advantageous to have some messages that display data used or collected. The **Trace** command is useful for this purpose. Messages can be displayed to the user to explain the progress of the test. In addition, good programming practices include taking groups of steps that are frequently executed together and combining them into functional blocks called procedures. In this example, a procedure called 'StopTest' is used to clean up the test before exiting. The StopTest procedure (not shown) includes steps to disconnect the Composer test from the IxN2X by using the **StopSession** command and the **Trace** command to display to the user the reason for stopping the test.



	Command Type	Session	Return Variable	Command String
1	StartSession	IxN2X	session_info	IxN2X, Session Start n2xHostname=localhost n2xAppVersion=6.12 n2xSessionLabel=TC_PPPoE n2xReuseTestSession=yes
2	Execute	IxN2X		Session LoadConfigurationFile n2xCsrFilePath="C:/Documents and Settings/whand/My Documents/My N2X Configuration/pppoeclientwithtraffic.xml" targetPortNameList="101/1 101/2"
3	Execute	IxN2X		Link RunArp portList=ALL enableSelfPaced=yes
4	Sleep			00:00:15.000
5	Execute	IxN2X		Protocol PppAction portList="101/1" protocolEntity="PPP over Ethernet Client" action="Enable" filter=contains pattern="<anything>" itemList="PPP over Ethernet Client 1"
6	Sleep			00:00:15.000
7	Execute	IxN2X		Protocol PppAction portList=101/1 protocolEntity="{PPP over Ethernet Client}" action="{Open connections}" filter=contains pattern="<anything>" itemList="{PPP over Ethernet Client 1}"
8	Sleep			00:00:15.000
9	Execute	IxN2X	PPP_Stats	Statistics GetProtocolStateStatistics portList=101/1 protocol=PPPOE_CLIENT
10	GetValue		Established_sessions	PPP_Stats Established Port {101/1}
11	If			$Established_sessions != 100
12	RunProc...			Stop_Test "Test Failed: Some of the sessions are not up Expected Sessions: 100 Actual Sessions : $Established_sessions"
13	Return			0
14	EndIf			
15	Execute	IxN2X		Traffic UpdateMeshStreamGroups meshList="{TrafficMesh 1}"
16	Sleep			00:00:15.000
17	Execute	IxN2X		Statistics ConfigurePortView view=Stats portList="101/1 101/2" statisticsList="AGT_TEST_PACKETS_RECEIVED AGT_TEST_PACKETS_TRANSMITTED"
18	Execute	IxN2X		Traffic StartTrafficEngine maxWaitSeconds=10
19	Sleep			00:00:30.000
20	Execute	IxN2X		Traffic StopTrafficEngine maxWaitSeconds=10
21	Execute	IxN2X	Results	Statistics GetView view=Stats statisticsList="TEST_PACKETS_RECEIVED TEST_PACKETS_TRANSMITTED"
22	GetValue		packets_received	Results test_packets_received Port {101/2}
23	GetValue		packets_transmitted	Results test_packets_transmitted Port {101/1}
24	Trace			Packets Received = $packets_received Packets Transmitted = $packets_transmitted
25	If			$packets_received < $packets_transmitted
26	RunProc...			Stop_Test "Test Failed: Packets were lost Packets Received = $packets_received Packets Transmitted = $packets_transmitted"
27	Return			0
28	EndIf			
29	RunProced...			Stop_Test "Test Passed: Packets Received = $packets_received Packets Transmitted = $packets_transmitted"
30	Return			1

Figure 76. Composer PPPoE Feature Test Script with IxN2X session commands

Contact Ixia

Corporate Headquarters **Ixia Worldwide Headquarters** **26601 W. Agoura Rd.** **Calabasas, CA 91302** **USA** **+1 877 FOR IXIA (877 367 4942)** **+1 818 871 1800 (International)** **(FAX) +1 818 871 1805** **sales@ixiacom.com**	**Web site: www.ixiacom.com** **General: info@ixiacom.com** **Investor Relations: ir@ixiacom.com** **Training: training@ixiacom.com** **Support: support@ixiacom.com** **+1 877 367 4942** **+1 818 871 1800 Option 1 (outside USA)** **online support form:** **http://www.ixiacom.com/support/inquiry/**
EMEA **Ixia Technologies Europe Limited** **Clarion House, Norreys Drive** **Maiden Head SL6 4FL** **United Kingdom** **+44 1628 408750** **FAX +44 1628 639916** **VAT No. GB502006125** **salesemea@ixiacom.com**	**Renewals:** renewals-emea@ixiacom.com **Support:** support-emea@ixiacom.com **+44 1628 408750** online support form: http://www.ixiacom.com/support/inquiry/?location=emea
Ixia Asia Pacific Headquarters **21 Serangoon North Avenue 5** **#04-01** **Singapore 5584864** **+65.6332.0125** **FAX +65.6332.0127** **Support-Field-Asia-Pacific@ixiacom.com**	**Support:** Support-Field-Asia-Pacific@ixiacom.com +1 818 871 1800 (Option 1) online support form: http://www.ixiacom.com/support/inquiry/