Open2Test Test Automation Framework Tips and Tricks for TestPartner

Version 1.0

September 2009

TABLE OF CONTENTS

1. Introduction

1.1. Purpose

This “Tips and Tricks” document provides an overview of handling frequently encountered scripting problems and some valuable do’s and don’ts to maximize the productivity of the Open Source Test Automation Framework

1.2. Scope

This document is just for understanding and reference. There might be a need to tweak the solutions provided before implementation, depending on the actual scenario in hand.

This document requires prior knowledge and working experience with the Open Source Test Automation Framework. For understanding the keywords and syntaxes, please refer to the “Scripting Standards” document.

2. Conditions

Conditions are basically implemented when we need a specific code to be executed only when a condition is satisfied. For example, we would want a 'Pass' report to be sent to the log if a variable 'A' is equal to variable 'B', and a 'Fail' report otherwise.

Syntax – r |condition|<Var A>;<condition>;<Var B>|Start ROW;End ROW

2.1. Problem Scenarios

At times, you may encounter the following two problem scenarios during the implementation of conditions using the Open Source Test Automation Framework.

2.1.1. Handling nested conditions

At times, the flow might not be restricted to just one condition but instead require multiple conditions before code is executed. In other words, there are times when nested conditions might be required. In such cases, the second condition should be placed after the first condition and should fall within the range of the first condition.

The following example in **'Table 1'** shows a simple way to do this. The first condition at line 10 has the start and end rows as 11 and 13 respectively. The second condition is placed at line 11 and has the start row as line 12 where the third condition is placed. Therefore, if the first condition is satisfied, only then will the control go to the second condition. If the second condition is satisfied, the control would go to the third condition. Thus, during actual execution, all the three conditions would have to be satisfied before the code at line 13 is executed.

Line	A	B	C	D	Remarks
10	r	Condition	VarA;equals;VarB	11;13	First condition
11	r	Condition	VarB;equals;VarC	12;13	Second condition
12	r	Condition	VarD;greaterthan;VarE	13;13	Third condition
13	r	Report	Pass;Expected result :: Actual Result		

Table 1: Nested Conditions

2.1.2. Changing start rows and end rows during script maintenance

One of the major problems that often goes unnoticed during script development is the use of hard-coded values to specify the start and end rows of any condition. The row numbers pose a significant problem during maintenance when changes are implemented in the script that involves the addition and deletion of rows. Whenever a row is added or deleted, the start and end rows of a condition no longer hold true and thus need to be updated. This results in wasted time when the number of conditions is high.

To solve this problem, simple Excel functions can be used to specify the start and end rows. If the **ROW()** function available in Excel is used to specify the start and end rows, the line numbers get updated automatically whenever a new row is added or deleted in the script.

In the script below in **'Table 2,'** for the '1st Condition,' the formula used to specify the start and end rows (i.e., 11;13 is **"=ROW()+1&";"&ROW(A13)")**.

The ROW() function returns the row number of a particular reference cell. If no parameter is passed, then it returns the current row number. The formula used can be split into three parts for a better understanding:

1. **ROW()+1:** Generally all the conditions have the start row as the very next line. Here we are fetching the current row by 'ROW()' and then adding '1' to get the next row number.

2. **&";"&:** Since we have a semicolon ';' as the separator between the start and end rows, we have to concatenate it with the help of '&' operator in Excel.

3. **ROW(A13):** Here we are using a reference to get the end row. In the example shown, row (A13) returns the value '13.' In case a line is inserted between rows 10 and 11 and 11 and 12, the end row would automatically be incremented by 1.

Line	A	B	C	D	(Formula to be used)
10	r	Condition	VarA;equals;VarB	11;13	=ROW()+1&";"&ROW(A13)
11	r	Condition	VarB;equals;VarC	12;13	
12	r	Condition	VarD;greaterthan;VarE	13;13	
13	r	Report	Pass; Act Res:: Exp Res		

Table 2: Usage of Excel's ROW Function

This formula works with the help of references. Hence, the end row to which there is a specific reference should not be deleted. It can be modified and any number of rows can be deleted or added in between, but the end row should not be deleted.

3. BaseState

It is very important that the automation scripts developed should not have any dependency on the state the application is in. It should be able to execute from any screen in the application. Why is this important? When these scripts run in a suite and a script fails, the following script might not find the application in the home screen. In such circumstances, the second script should not fail because it did not find the application in the required state.

The best way to resolve this is to have code in place that would take the application to the main screen before beginning the execution of any script. One of the ways of doing this is to use a Reusable Action and call this Reusable Action at the first and last line of each script. For best results, this code can also be included in the error-handling functions.

The example below in **'Table 3'** shows the call to BaseState in the first and last lines of the script. The Base State function can be associated with certain parameters if required for additional flexibility.

Example:

Line	A	B	C	D	Remarks
1	r	Callaction	RA_BaseState	Param	**Base State function call.**
2	r	Keyword	Keyword	Keyword	**Script Body**
10	r	Keyword	Keyword	Keyword	
15	r	Callaction	RA_BaseState	Param	**Base State function call.**

Table 3: Base State

4. Message Box

Maintenance is an integral part of the automation lifecycle. Once we have developed the scripts, we have to maintain them to keep the script updated and in sync with the application.

The "message box" functionality proves to be very handy for maintaining scripts. You can use the "MsgBox" keyword to invoke a message box with the value of the dynamic variables during the execution of the script. This, to some extent, helps in pinpointing the error points during execution.

In '**Table 4'** at line 10, 'Text' property of a 'textbox' object with the name 'Name' is stored in a variable 'strValue' for later use at line 12, where it is being checked to be equal to "Smith." During maintenance, we can actually see what is getting stored in the variable '#strValue' by inserting a simple 'msgbox' keyword with '#strValue' as the argument.

Line	A	B	C	D	Remarks
10	r	Storevalue	Textbox;Name	Text:strValue	
11	r	Msgbox	#strValue		**'Msgbox' used.**
12	r	checkcondition	#strValue;equals;Smith	13;13	
13	r	Report	Pass; Act result :: Exp result		

Table 4: Message Box 'msgbox' Keyword

5. Loops

Loops prove to be a very useful tool when we have to execute code repeatedly for a given set of data. This section describes how to use the looping keywords in the Open Source Test Automation Framework to construct a loop.

Scenario: In this scenario, we have to create a set of users for the application under test. This is a repetitive job. The loop keyword of the Open Source Test Automation Framework has been designed to handle such scenarios.

To use the loop keyword, we have to place the data in one of the columns of the 'Data sheet'. For example, in Table 5 the names of the users are to be entered in a column called 'NAME'.

	NAME	B	C
1	User 1		
2	User 2		
3	User 3		

Table 5: Data Sheet, Mapped using Active Data

As shown in the example below, at line 10 we have the loop keyword that has the start and end rows (11;13) as the argument. Therefore, lines 11 and 13 would serve as the body of the loop. At line 11, we are setting the context and in line 12 we are entering the user name. Here, the value has to be retrieved from the 'NAME' column of the 'Data sheet.' Hence, we have used 'set:dt_NAME'. By default, the number of iterations of the loop is defined by the number of start and end rows indicated in row 10, column C. In the current example, the loop would continue three times, one loop for 11, 12, and 13 as indicated by 11;13.

If we want to run it two times, then it could be done by inserting '2' in the 'D' column in line 10.

Line	A	B	C	D	Remarks
10	r	Loop	11;13		**Loop Keyword**
11	r	Context	Window;Create_User		
12	r	Perform	Textbox;UserName	Set:dt_NAME	**Setting value from col 'NAME' of 'Data sheet' of TestPartner.**
13	r	Perform	Button;Submit		

Table 6: Loops

6. Use of Wildcards

TestPartner supports the use of wildcards in objects. If the object attach names are changing dynamically, to identify these kinds of objects we can use wildcards or regular expressions in combination with the attach name properties.

Example:

Attach Name for HTML Browser = “Browser1 SampleApplication”

Properties for HTML Browser: Caption=Browser1 Name=Browser1

If attach Name changing to “Browser2 SampleApplication”, or “Browser3 SampleApplication” in runtime, then we can identify using following properties with wildcards.

Caption=Browser*

Caption=?rowser*

Name=Browser?

Name=Browse?*

Line	A	B	C	D	Remarks
14	r	Perform	browser;Caption=Browser*	attach	Attach browser
15	r	Perform	browser;Caption=Browser*	close	Close browser

Table 7: Use of Wildcards