**PowerShell Cmdlets Test Cases Auto Generation**

LiPing Yu, Tao Zhou, Rong Zhou

Abstract:

In PowerShell Cmdlet feature testing, one important testing area is to test the parameter combinations for each Cmdlet. For a Cmdlet with 6 parameters, the largest number of test cases is 360. When considering about the mandatory/optional, named/position/switch parameters, the condition is more complex. It’s impossible to design all these test cases manually. This paper describes a test framework which generates parameter combination test cases automatically for PowerShell Cmdlets. It is easy to implement test cases and even update test cases using this PowerShell testing framework.

1. Introduction

When testing PowerShell Cmdlets, it is always a headache to make sure all parameter combinations are covered. The number of parameter combination grows rapidly with the increase of parameters. If negative cases need to be covered, the total number of combinations will be tremendous. It is easy to bring errors to the test cases if the tester doesn’t design test cases carefully. So a test framework is designed to generate all the test cases automatically.

shows the overall architecture of the framework.

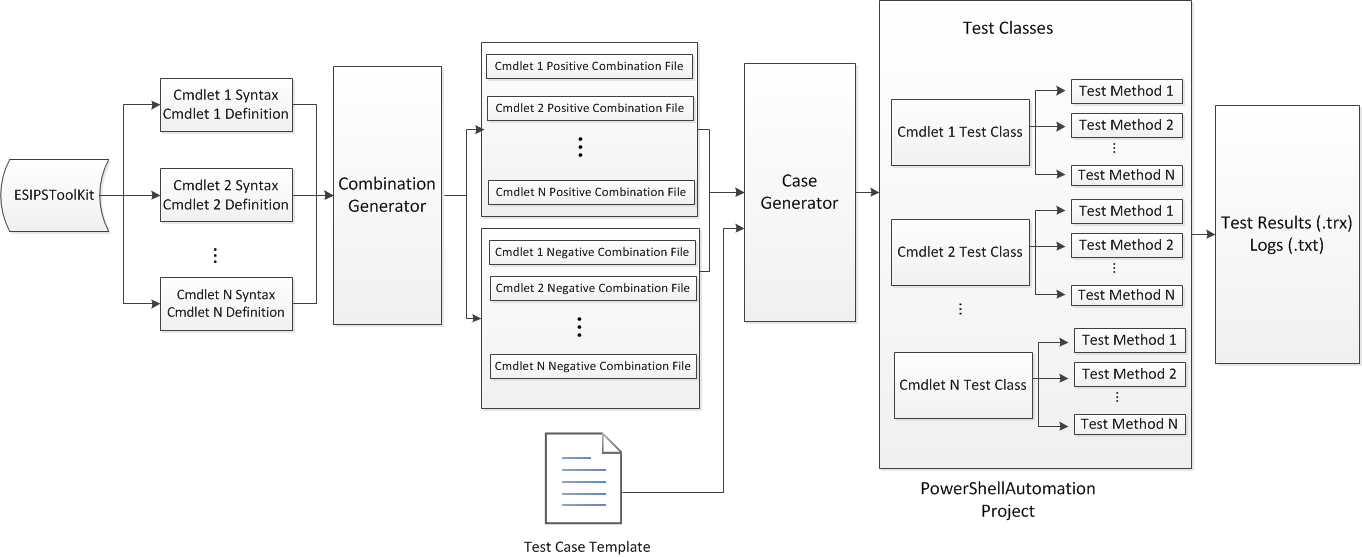


Figure 1

There’re three key components in this testing framework: the Combination Generator, the case generator and the PowerShellAutomation project. The PowerShellAutomation project is a C# project which is created based on Microsoft Visual Studio Unit Testing Framework. It’s the application where testers run and verify test cases. It can be customized according to the requirements for the testing. The Combination Generator is an application to generate Cmdlets combinations and the Case Generator is an application to generate test methods/test cases. This paper will mainly cover these two components and introduces how PowerShell Cmdlet test cases are automatically generated.

1. Combination Generator Model

The syntax of a Cmdlet usually contains several parameters including mandatory and optional ones. Thus one Cmdlet may have multiple usages due to different combinations of different parameters. For each Cmdlet, there are two types of combinations, one is positive combination, and the other is negative combination. If a Cmdlet has **m** mandatory parameters and **n** optional parameters (assuming each parameter has a value), the algorithm of generating a positive combination is:

1. Select all **m** mandatory parameters.
2. Select **k** parameters from optional parameters **()**
3. Combine selected parameters in step a and b.

The total count of the positive combination is:

And the algorithm of generating a negative combination is:

1. Generate a positive combination
2. Remove a key or value which is mandatory from each positive combination

The total count of the negative combination is:

For example, the cmdlet New-EmcLun has the following syntax in the original version:

|  |
| --- |
| New-EmcLun -Pool <StoragePool> [-Name <String>] -Capacity <UInt64> [-Thin] [-Description <String>] [-HostbusAdapter <HostBusAdapter>] [-Silent] [<CommonParameters>] |

It has 2 mandatory parameters: Pool and Capacity; and it also has 5 optional parameters: Name, ApplicationHint, Thin, Description, HostbusAdapter and Silent. So there are totally 32 positive combinations of parameters and () =158 negative combinations.

In the later version, the syntax of the Cmdlet New-EmcLun is changed:

|  |
| --- |
| New-EmcLun -Pool <StoragePool> [-Name <String>] -Capacity <UInt64> [-ApplicationHint <String>] [-Thin] [-Description <String>] [-HostbusAdapter <HostBusAdapter>] [-Silent] [<CommonParameters>] |

An optional parameter is added and the total count of positive combinations is =64, and the total negative combination is) =298.

We can see that one optional parameter can bring us a lot of effort if we generate the combinations manually.

Since the parameters of a Cmdlet usually change during the development phase, it is really an unimaginable hard work to generate and maintain all the combinations of all the Cmdlets manually. So the Combination Generator Model is coming.

The Combination Generator queries usage from a PowerShell module like PowerShell ESIPSToolKit which is located on the test machine to get command syntax and definition and generates the appropriate combination files. The only input required for the Combination Generator is the Cmdlet syntax and definition which can be retrieved from “Get-Command” command. So the Combination Generation can be applied to any PowerShell Cmdlets.

is the framework of Combination Generator Model.



Figure 2

The Combination Generator will generate 2 files for each Cmdlet according to the PowerShell command syntax and definition. One is positive test case combination file which contains all positive combinations of the Cmdlet. The other is negative test case combination file which contains all negative test case combinations. All files are plain text files.

Here is an example of Cmdlet “New-EmcLun”. The parameters “Pool” and “Capacity” are mandatory, and others are optional. is the positive test case combination file of Cmdlet “New-EmcLun” and is the negative test case combination file.



Figure 3



Figure 4

1. Case Generator Model

One Cmdlet may have hundreds of parameter combinations. When the parameters change, the combinations will change dramatically which is impossible to maintain the test cases manually. The Case Generator is designed to automatically generate the test cases according to the combination of Cmdlet. When the parameter combinations change, the Case Generator will update the test cases automatically.

Case Generator is designed based on Visual Studio Unit Testing Framework [1]. The Unit Testing Framework supports unit testing in Visual Studio. It uses attribute [TestMethod] to identify a test method. Test cases are implemented in test methods. All test methods are members of test classes which are decorated with [TestClass()]. The Case Generator gets parameter combinations from combination files and generates test method for each combination. The test methods generated for the combinations of one Cmdlet will be inserted to a test class and saved in the two files. One file contains the test methods for positive test cases. And the other contains the test methods for negative test cases.

Unit Test Framework will treat the TestMethod attribute marked method as test case, so if the file contains test methods is added into project, the test cases are created from test methods. The test cases will be list in test view window of Visual Studio.

To use the test method files, you should include [Microsoft.VisualStudio.TestTools.UnitTesting](http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting(v=vs.80).aspx) namespace in your test automation project and add these files to the project. The test cases will be listed in Test View Window. Tester can execute test cases easily from Test View Window or using mstest.exe CLI.

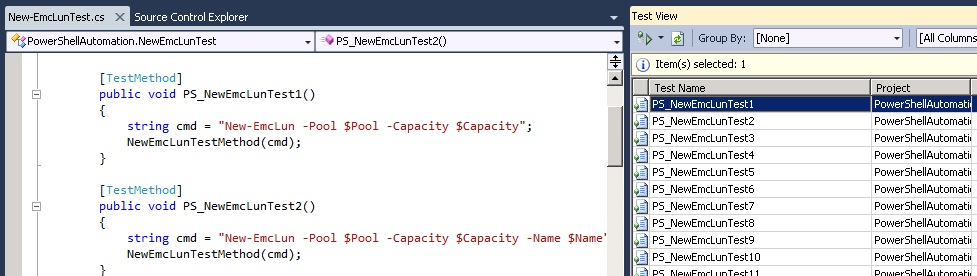


Figure 5

If a Cmdlet changes, the parameter combination will be updated by Combination Generator. And Case Generator will generate test methods based on new parameter combinations. In this way, the test cases implemented in test methods are always kept updated.

shows how Case Generator works. There are template files which contain the common code structure of a test case. The Case Generator will read the template files and combination files of the Cmdlets and generate test cases.

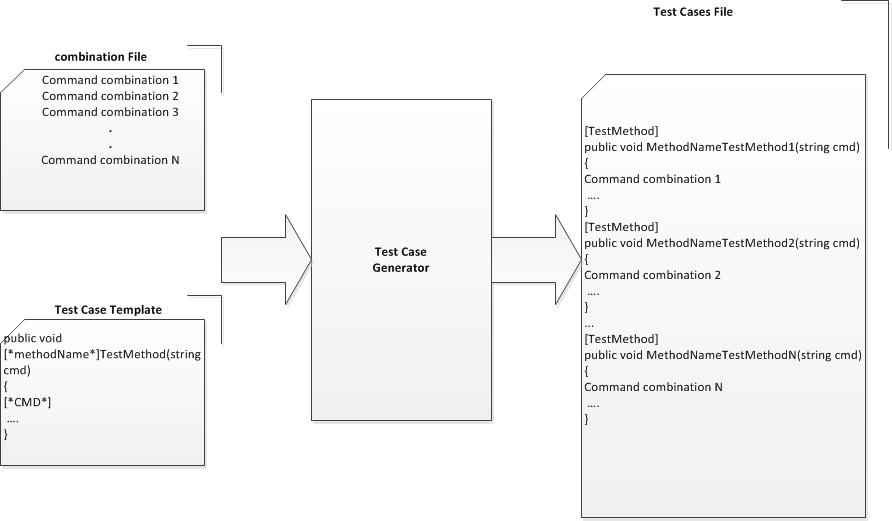


Figure 6

For example, in our product, there’s a Cmdlet named New-EmcLun. shows how Case Generator generates test method files for this Cmdlet. The Case Generator will use two template files: one is Template.cs, a test class template, and the other is TestMethodTemplate.cs, a test method template. The string decorated with [\* \*] will be replaced by appropriated value. The [\*testName\*]Test[\*para\*] is the test method for each combination which implements test cases marked with [TestMethod] attribute. In the final test method files, [\*testName\*] is replaced by Cmdlet Name “New-EmcLun”, [\*para\*] is the test case ID and [\*cmd\*] is replaced by combination string. The command combination string is read from Combination files.

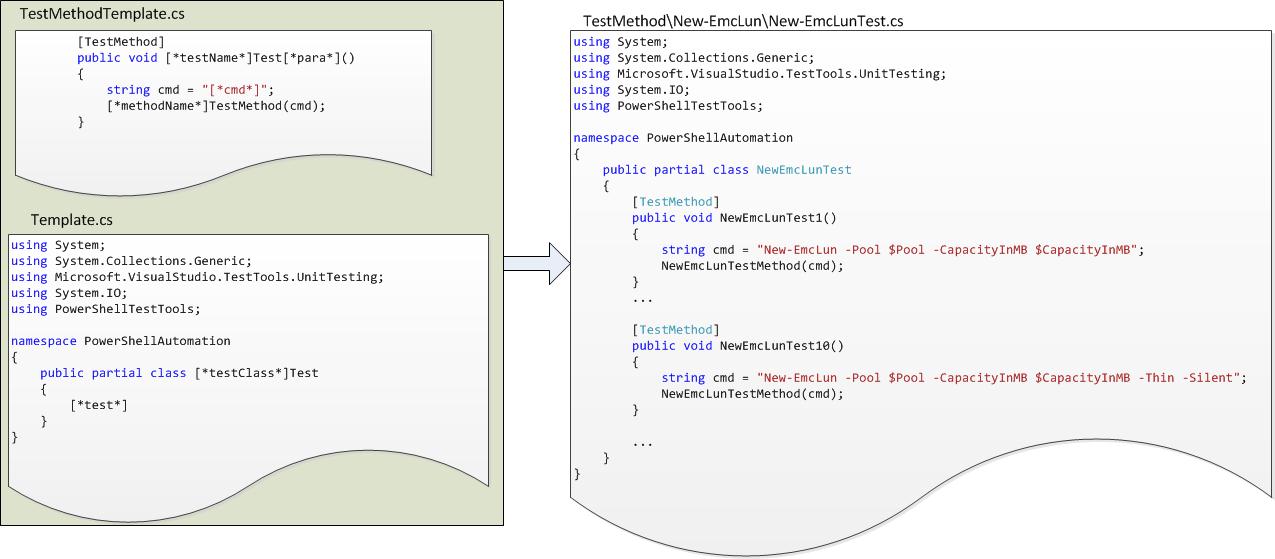


Figure 7

1. Application

This testing framework is applied to test ESG ESI PowerShell module. The framework automatically generates 10586 test cases for 87 Cmdlets, and 71426 lines of code. Assuming a QA can design 300 test cases and implemented 200 lines of code a day. It saved 35 days in test cases design and 357 days in test automation coding. The average code coverage is increased to 81.12%.

1. Summary

The testing framework described in this paper shows a way to generate test cases automatically for PowerShell Cmdlets. It releases QA engineers from boring, time-consuming work of parameter combination test cases design. The benefits of the framework are listed below:

* This is a generalized framework. The only input required is the Cmdlet syntax and definition which can be retrieved from “Get-Command” command. So when you know the name of the PowerShell module which you’re going to test, Combination Generator will retrieve all Cmdlets of the module. It will analyze the parameters of each Cmdlet and combine the parameters properly. So this framework is not designed for any specific PowerShell module. It’s designed for all PowerShell modules.
* Tester can take advantage of test combinations files generated by Combination Generator even they don’t want to use Case Generator. The Combination Generator can work separately from Case Generator. All the combination files are saved in plain text files. So a tester can use the test combination files to design test cases and then use any programming scripts/languages, i.e., PowerShell, Perl, C#, to execute the test cases. This makes the framework more flexible.
* This framework ensures all the parameter combinations are covered. It improves code coverage rate. Since all test cases are generated automatically, it prevents testers from manual errors when designing test cases.
* The framework is easy to scale out. In a real project, Cmdlets change from time to time during the early stage of product development. Testers have to refine their cases accordingly which costs extra efforts. Using this framework, it’s not a problem anymore. The framework itself will take care of the changes of Cmdlets. For example, when new Cmdlets are added, the framework will be aware of the new Cmdlet and generate test cases for it automatically. The more exciting feature is when the parameters of a Cmdlet are changed, the framework will also update the test cases accordingly.

1. Future Improvement

Current the test cases generated by the framework only cover the combination for parameters. It doesn’t cover the generation for parameter value. Engineers have to either input the value in a configuration file or generate the value during test case implementation. One reason is the test data selection depends on different features. To define valid test data for a test case, QA has to be very familiar with the features they’re testing. A good test data definition can help to improve test coverage and catch more bugs. Next step we’re going to do is to analyze the basic rules/strategies for parameter value selection/generation [2]. If parameter values can be simulated automatically, it will save more human efforts. This is especially meaningful when handling negative test cases.

References

[1] Unit Testing Framework <http://msdn.microsoft.com/en-us/library/ms243147(v=vs.80).aspx>

[2] Combination Testing Strategies: A Survey. Mats Grindal, Jeff Offutt, Sten F. Andler.