Automated Testing of Cloud Applications

Linghao Zhang, Tao Xie, Nikolai Tillmann,
Peli de Halleux, Xiaoxing Ma, Jian lv

{lzhang25, txie}@ncsu.edu, {nikolait, jhalleux}@microsoft.com, {xxm, lj}@nju.edu.cn

Abstract: Recently, cloud computing platforms, such as Microsoft Azure, are available to provide convenient infrastructures such that cloud applications could conduct cloud and data-intensive computing. To ensure high quality of cloud applications under development, developer testing (also referred to as unit testing) could be used. Generally, manual developer testing is time consuming and labor intensive. To reduce the manual efforts, developers could employ automated test generation tools. However, the behavior of a unit in a cloud application is often dependent on the state of the cloud environment. Applying an automated test generation tool faces the challenge to generate various cloud states for achieving effective testing, such as achieving high structural coverage of the cloud application. To address this challenge, we propose an approach to (1) use parameterized mock objects to mimic the behavior of the environment and, (2) apply dynamic symbolic execution (DSE), a state-of-the-art automated test generation technique, to both generate test inputs and mock cloud states to achieve high structural coverage. We apply our approach on some open-source Azure cloud applications. The result shows that our approach automatically generates test inputs and mocks cloud states to achieve high structural coverage.

Keywords

Cloud Computing, Software Testing, Dynamic Symbolic Execution, Parameterized Mock Objects

Introduction

Cloud computing has become a new computing paradigm where the cloud could both provide virtualized hardware and software resources that are hosted remotely and provide a use-on-demand service model. One typical service model of cloud computing is Platform as a Service (PaaS). Such cloud platform services, such as the Microsoft Azure platform [1], provide convenient infrastructures for conducting cloud and data-intensive computing. After deploying an application to the cloud, one can access and manage it from anywhere using a client application, such as an internet browser, rather than running or storing the application locally. For example, a typical Microsoft Azure cloud application consists of web roles, i.e., typical web-service (client-interfacing) processes deployed on Internet

Information Services (IIS) hosts, and worker roles, i.e., background-processing (such as computational and data management) processes deployed on system hosts. Web roles and worker roles communicate with each other via queues. Both web roles and worker roles access storage services in the Azure cloud.

To ensure high quality of cloud applications under development, developer testing (also referred as to unit testing) could be used. Generally, testing a unit with all possible inputs is impossible since the input space is too large or even infinite. Therefore, we need a criterion to decide which test inputs to use and when to stop testing. Coverage criteria (such as structural code coverage) could be used for such purposes and effective use of coverage criteria makes it more likely that faults could be revealed [2]. Among different coverage criteria, structural code coverage is the most commonly used one. Although full structural code coverage cannot guarantee fault-free software, a code coverage report for showing less than 100% code coverage indicates the inadequacy of the existing test cases, e.g., if a faulty statement is not covered by all the execution of any existing test case, then this fault can never be revealed. To achieve high structural coverage, test cases can be written manually; however, manual writing test cases is labor-intensive. To reduce the manually efforts, testers or developers could employ automated test generation tools that automatically generate test inputs to achieve high structural coverage, such as Dynamic Symbolic Execution (DSE) [3] (also called concolic testing [4], a state-of-art structural testing technique) based tools.

In order to test a cloud application before deployment, various desktop-based cloud-environment emulators, such as Microsoft Azure Compute and Storage Emulators [5], enable developers to run and test their cloud applications locally. To directly manipulate some parts of the cloud environment (either the local emulated environment or the remote real environment), we could leverage some cloud management tools, such as the Cloud Storage Studio for Microsoft Azure. Since cloud applications are actually cloud-environment-dependent applications, the behavior of a unit under test in a cloud application is dependent on the input to the unit as well as the state of the cloud environment. Automated test generation tools would fail to generate high-covering test inputs because these tools generally lack knowledge on how to generate a required state of the cloud, or even cannot control the state of the cloud. Using Mocking techniques to isolate the application unit under test could alleviate those issues. Existing mocking frameworks, such as NMock [6] [6] and Moles [7], use lightweight simulation/implementation to replace the application unit's interactions with the cloud environment. Particularly, the mocked cloud API methods provide some default or customized return values without reflecting the logic or the state consistency of the actually ones, resulting false warnings. In addition, developers still need to manually provide the expected return values of each API method in order to cover some particular parts of the unit under test.

To address the limitation of existing mocking techniques, we propose to use a parameterized mock cloud [8] to simulate various possible return values automatically. This mock cloud could enable automated test generation tools to explore all the feasible execution paths, and later use the information collected from different paths to generate test inputs and a mock cloud state to achieve high structural coverage. In addition, to reduce false warnings, we implement our cloud as a stateful cloud such that it replicates the effect of cloud API methods by performing the same operations on itself.

Background

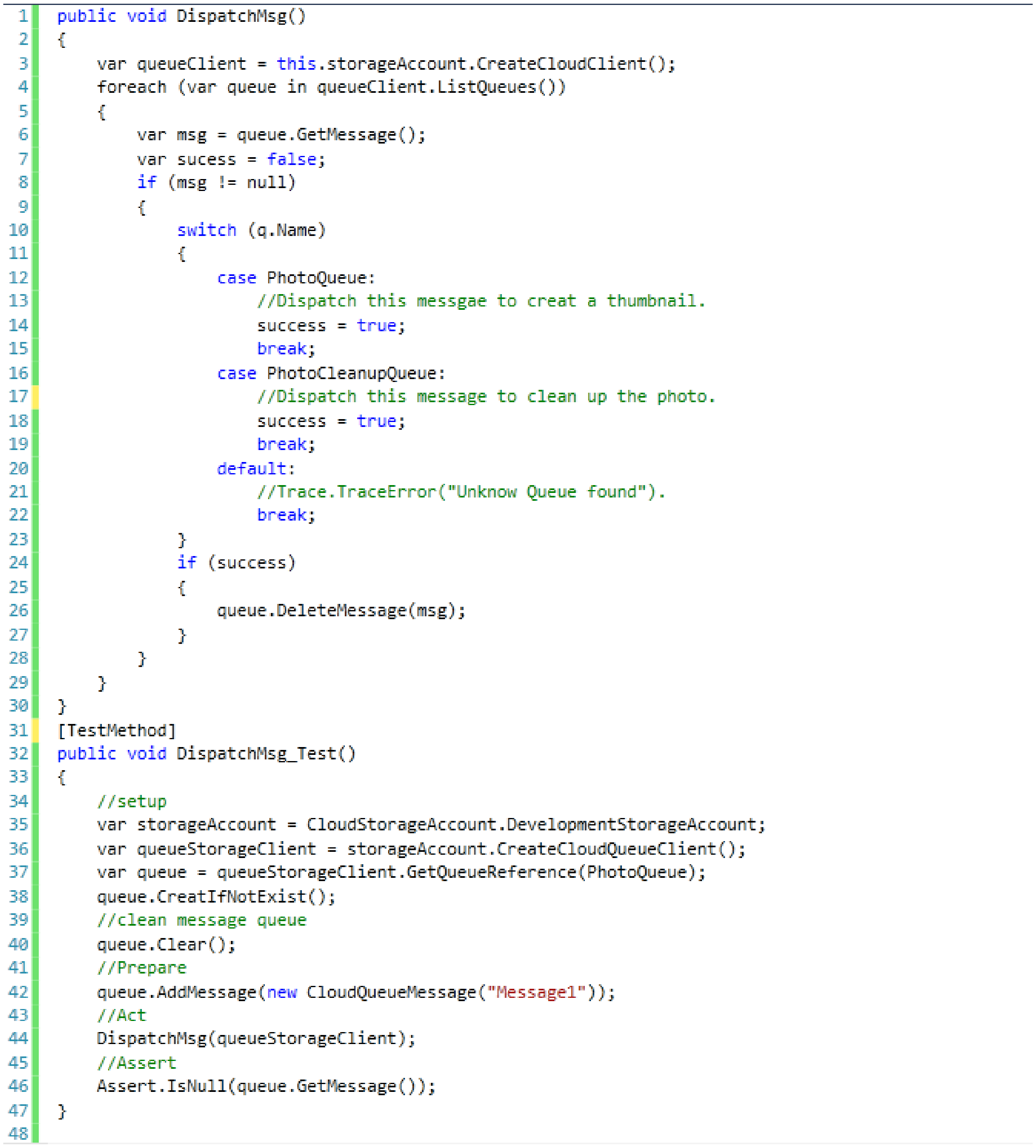
DSE is a constraint-solving-based technique that combines concrete execution with symbolic execution. It could be used in both test generation and program analysis. DSE automatically explores the space of program paths while incrementally increasing the code coverage such as block or branch coverage. DSE initially executes the program under test with a default or random test input. When encountering a branch statement, DSE collects the symbolic constraints on the taken branch of the statement. The conjunction of all symbolic constraints along an executed path is called path condition, which represents an equivalence class of concrete input values that take the same path. By flipping a taken branch in the executed path, DSE constructs a new path that shares the prefix to the taken branch with the executed path, but takes a different branch at the flipped point. Pex [7] is an automatic white-box test generation tool for .NET, based on dynamic symbolic execution. Pex has been integrated into Microsoft Visual Studio as an add-in. A key methodology that Pex supports is parameterized unit testing [9], which generalizes unit testing by allowing unit tests to have parameters.

Empirical Investigations

We surveyed open source Azure projects available from *Codeplex* and *Google Code* (21 projects in total, whose details can be found on our project web site [10]). Among them, 5 projects include unit tests. We manually investigate the unit test result of the *Lokad.Cloud* project because there exist 3 classes that heavily interact with the cloud environment and a number of test cases written to test almost every method in those 3 classes. The unit tests achieved 80% block coverage for class *BlobStorageProvider*, 79% for class *QueueStorageProvider*, and 93% for class *TableStorageProvider*. We carefully inspect the not-covered blocks in these 3 classes and find out that there are four reasons causing a block not covered: (1) covering it requires a specific cloud state; (2) covering it requires a specific program input; (3) the method that it belongs to is not executed by any test case; (4) covering it depends on other business logic. In summary, 78% (111/141) blocks are not covered because the existing test cases fail to provide either specific cloud states or program inputs. In addition, most test cases, which are written for testing a unit that interacts with the cloud environment, begin with a manual step of preparing environment setup and these test cases must run against a local cloud environment simulator. Different execution paths of a unit under test require different combinations of program inputs and cloud states, and developers may miss some combinations when writing test cases (including setting up a cloud state).

Testing Challenge

We next illustrate the testing challenge with an example shown in Figure 1. The code snippet is a simplified method with a unit test from an open source project *PhluffyFotos* [11]. The method *DispatchMsg* first acquires a *CloudQueueClient* from the *StorageAccount* at Line 3 and gets a list of existing *MessageQueues* at Line 4. Then this method fetches one message from each queue at Line 6 and dispatches the message to another message-processing method according to the type of each queue at Lines 10-23. The flag *success* is assigned to be true if the message has been successfully dispatched and processed at Lines 14 and 17. Finally, this method deletes the message at Line 26 if the flag *success* is true.



```
 1  public void DispatchMsg()
 2  {
 3      var queueClient = this.storageAccount.CreateCloudClient();
 4      foreach (var queue in queueClient.ListQueues())
 5      {
 6          var msg = queue.GetMessage();
 7          var sucess = false;
 8          if (msg != null)
 9          {
10              switch (q.Name)
11              {
12                  case PhotoQueue:
13                      //Dispatch this messgae to creat a thumbnail.
14                      success = true;
15                      break;
16                  case PhotoCleanupQueue:
17                      //Dispatch this message to clean up the photo.
18                      success = true;
19                      break;
20                  default:
21                      //Trace.TraceError("Unknow Queue found").
22                      break;
23              }
24              if (success)
25              {
26                  queue.DeleteMessage(msg);
27              }
28          }
29      }
30  }
31  [TestMethod]
32  public void DispatchMsg_Test()
33  {
34      //setup
35      var storageAccount = CloudStorageAccount.DevelopmentStorageAccount;
36      var queueStorageClient = storageAccount.CreateCloudQueueClient();
37      var queue = queueStorageClient.GetQueueReference(PhotoQueue);
38      queue.CreatIfNotExist();
39      //clean message queue
40      queue.Clear();
41      //Prepare
42      queue.AddMessage(new CloudQueueMessage("Message1"));
43      //Act
44      DispatchMsg(queueStorageClient);
45      //Assert
46      Assert.IsNull(queue.GetMessage());
47  }
48
```

Figure 1. A method under test with a unit test in the *PhluffyFotos* project.

If developers want to write test cases to test this method, they need to first clean up the cloud to avoid that the old cloud state may affect the test result, and then prepare an appropriate cloud state before executing this method. An illustrative manually written test case at Lines 31-47 first gets a reference of a *CloudQueue "PhotoQueue"* at Line 37 and cleans all the messages in this queue at Line 40, and then executes this method at Line 44 after inserting a new message into the queue at Line 42. The assertion at Line 46 is to check whether the message has been deleted or not. However, if we want to cover all the branches of this method, we need to provide various cloud states. In particular, to cover the true branch at Line 8, at least one queue should be empty; to cover the true branch at Line 24, at least one of the *PhotoQueue* or *PhotoCleanupQueue* should exist with at least one message in the queue. For this relative simple method under test, developers already need some effort to construct the cloud state. Some branches of a more complex method or unit test may require some specific cloud states that cannot easily be constructed manually due to the complex execution logic.

Automated test generation tools usually require executing all the cloud-related API methods (by instrumenting these methods) to collect necessary information for test generation. Particularly, tools, such as Pex, use symbolic execution to track how the value returned by a cloud-related API method is used. Depending on the subsequent branching conditions on the returned value, these tools execute the unit under test multiple times, trying different return values to explore new execution paths. However, directly applying Pex would fail due to the testability issue because the cloud-related API methods are depending on the cloud environment that Pex cannot control.

Using mocking techniques, a mock object (with its mock API methods) can be generated automatically; however, it is still the responsibility of developers to simulate possible return values for each mock method. For example, developers manually provide a list of *CloudQueue* as the return value of the method *ListQueues()*. A mock object enables Pex to automatically generate various inputs and return values for the unit under test to explore different execution paths. However, such mocking techniques generally cannot reflect the changes of the cloud environment. For example, after the method *DeleteMessage(msg)* at Line 26 in Figure 1 is executed, the message *msg* should be deleted from the queue, and the return value of method *GetMessage()* at Line 46 should be null. If a mock object cannot capture this behavior, the method *GetMessage()* may return a non-null value even the method *DeleteMessage()* has been executed. Consequently, this test case fails in the assertion at Line 46, causing it a false warning.

Addressing Testing Challenge

To address the challenge of automated testing of cloud applications, we propose a new approach with a Parameterized Mock Cloud. Given a unit of a cloud application under test, our approach includes four parts: Cloud Mocking, Code Transformer, Test Generator, and Test Transformer.

Mocking the cloud

A simple or native implementation of a mock cloud environment generally cannot reflect the actual behavior of the real cloud environment, causing false warnings in the test results. We mainly implement a simulated mock cloud environment and provide mock cloud API methods that replicate the effect of the corresponding API methods on the real cloud environment by performing the same operations on the mock cloud environment. In particular, our mock cloud currently focuses on providing simulated Azure storage services and mocking the classes in the *Microsoft.WindowsAzure.StorageClient* namespace, which provides interactions with Microsoft Azure storage services. Microsoft Azure storage services provide three kinds of storage: Blob (abbreviation for Binary Large Object), which is used to store things such as images, documents, and videos; Table, which provides queryable structured storage that is composed of collections of entities and properties; Queue, which is used to transport messages between applications.

To implement such a mock cloud, we not only carefully read the API documents from MSDN but also read though many code examples from the investigated open source projects. Here, our mock cloud is implemented using a test-driven approach, where we mock different classes and functionalities incrementally by the demand of a unit under test rather than mocking the whole storage services all at once. The name of each mock class starts with "Mock", and ends with its original name. For example, the mock class for class *CloudQueue* is named as *MockCloudQueue* in our mock cloud. The name of each method is the same as the original one. We build up the three kinds of storage based on C# generic collections. Currently, we have mocked all the main classes and API methods in the three storage services. Queue storage is mocked using an instance of List<*MockCloudQueue*>, where each *MockCloudQueue* is mocked using an instance of List<*MockCkoudMessage*>. Blob storage is mocked using an instance of List<*MockContainer*> and each MockContainer is mocked using an instance of List<*MockIBlobItem*>. Table storage is mocked with a similar way.

Transforming Code Under Test

With a mock cloud, we execute a unit under test with the mock environment rather than the real cloud environment. Code Transformer redirects a unit under test to interact with our mock cloud environment. This process is done by pre-processing a unit under test. Specifically, if the target unit under test refers to Class *A* in the *Microsoft.WindowsAzure.StorageClient* namespace, this reference is redirected to class MockA; when a method *M* of Class *A* is invoked, this invocation is replaced by the simulated method in class *MockA.M*. Then, the processed unit under test would now interact with our mock cloud.

Generating Test Inputs and Cloud States

The Test Generator incorporates an automated test generation tool, Pex, to generate both test inputs and required cloud states for a unit under test. Specifically, Pex generates not only symbolic program inputs but also symbolic cloud states that include various storage items (such as container, blob, message, and queue) to be inserted into the mock cloud before the execution of the unit under test. Pex performs concrete execution on the unit under test with default or random values and performs symbolic execution to collect path constraints. By flipping a taken branch of the collected path constraints and solving the new constraints, Pex acquires a new program input and cloud that lead to a new execution path. In the end, Pex produces a final test suite where each test includes a test input and a cloud state. The algorithm for Queue storage state generation is shown in Figure 2.

We also add various constraints to ensure that Pex could choose a valid value for each field of a storage item. For example, if we test a cloud application using the *DevelopmentStorageAccount*, the Uri address for any blob container should be "`http://127.0.0.1:10000/devstoreaccount1/containerName`". Pex would choose only the name for each container, making the Uri address field valid. We use a similar algorithm to generate the blob storage states. But the algorithm to generate Table storage states is a little different. Practically, different types of entities can be stored in the same cloud table, but most open source projects use only one cloud table to store a particular type of entities. Therefore, we also restrict each *MockTable* to store only one type of entities. The algorithm for generating Table storage also requires the types of entities (an entity type is similar to a data schema but much simple) to be stored in each table. Such simplification enables Pex more easily to generate table storage states without losing much applicability.

```
1.  N ← Pex chooses the total number of MockCloudQueues (0 to MAX).
2.  QueueList ← Initiallize QueueStorage using an instance of List< MockCloudQueue > (N).
3.  for i from 0 to N
4.          MockCloudQueue_i ← Create a new instance of MockCloudQueue
5.          Pex chooses values for each field of MockCloudQueue_i
6.          M ← Pex choose the total number of MockCloudMessages (0 to MAX) to be inserted in to MockCloudQueue_i
7.          for j from 0 to M
8.              MockCloudMessages_ij ← Create a new instance of MockCloudMessage
9.              Pex chooses values for each field in MockCloudMessages_ij
10.             MockCloudQueue_i. MessageList.MessageAdd(MockCloudMessages_ij)
11.         end for
12.         QueueList.Add(MockCloudQueue_i)
13. end for
```

Figure 2. Algorithm for Queue storage state generation.

Transforming Generated Unit Tests

Testing the code under test with only the mock cloud environment is insufficient. To gain high confidence on the correctness of the code, testing the code with either the local emulated cloud environment or the real cloud environment is necessary. The Test Transformer transforms a generated unit test together with a cloud state into a general unit test. Specifically, the Test Transformer transforms a cloud state generated by the Test Generator to a sequence of real cloud API methods that could construct the same state as in the real cloud environment.

Testing Example with Mock Cloud

Now let us use the same example in Figure 1 to illustrate how our approach works. In the mock cloud queue storage, we use an instance of *List<MockCloudQueue>* named *MockQueueList* to store all the existing queues, and use an instance of *List<CloudMessage>* to represent a message queue. Initially, Pex would arbitrarily choose *N* (representing the total number of *MockQueueList*) to be 0, and there will exist no queue. So the execution would directly jump out of the loop at Line 4 and the path condition collected by Pex is "*N == 0 && N >= 0*". Then, Pex next tries to come up with a new cloud state by flipping the condition "*N == 0 && N >= 0*". By consulting the underlying constraint solver with the new path constraint, Pex chooses *N* to be *1* in the second run. Next, Pex would create a new *MockCloudQueue* and arbitrarily choose a name *"/0"* for this queue. Pex also chooses *M* (representing the total number *CloudMessages* in queue *"/0"*) to be *0*. Then the execution would take the false branch at Line 8 because there is no message in *MockCloudQueue* "/0". A new path condition collected by Pex in the second run is "*M == 0 && MockQueueList.name == "/0" && N == 1*"[1]. If the Depth-First-Search strategy is adopted, Pex would try to flip "*M == 0" to "M != 0*". In the third run, the path constraint is "*M != 0 && MockQueueList.name == "/0" && N == 1*" and one queue with one message is created. The execution would take the false branch at Line 12, adding a new constraint "*MockQueueList.name! = PhotoQueue*". Once Pex flips the constraint "*MockQueueList.name! = PhotoQueue*" in a certain run, Pex could create a new queue named "*PhotoQueue*", and then the true branch of Line 12 can be covered. As Pex keeps exploring the unit under test, finally, all the feasible paths/blocks can be covered with various cloud states. If the feasible paths are infinity or too many, Pex would stop at a certain termination condition. Finally, our approach generates nine cloud states that cover all the blocks in *Dispatch* method after 194 runs. The details of these generated cloud states can be found in our project web site [10].

As we have discussed earlier, a native implementation of a mock cloud can easily cause false warnings because it fails to reflect the actual behaviors of the real cloud environment. In contrast, our mock cloud can avoid false warnings by simulating the basic behavior of the cloud storage. The unit test shown in Figure 1 would pass using our mock cloud since the method "*queue.GetMessage()"* returns null. This return value would be the same when the unit is executed against the real cloud environment.

After Pex finishes exploring the unit under test with different cloud states, the Test Transformer transforms each cloud state to be a sequence of real cloud API methods. Suppose that one generated cloud state includes two queues named "*PhotoQueue*" and "*PhotoCleanupQueue",* and each queue contains one *CloudMessage "Msg1"*. Such cloud state would be translated into the following method sequence:

1. *var storageAccount = CloudStorageAccount.DevelopmentStorageAccount;*
2. *CloudQueueClient queueClient = storageAccount.CreatCloudQueueClient();*
3. *var queue1 = queueClient.GetQueueReference(Constants.PhotoQueue).CreatIfNotExist();*
4. *queue1.addMesssage(new CloudQueueMessage("Msg1"));*
5. *var queue2 = queueClient.GetQueueReference(Constants.PhotoCleanupQueue).CreatIfNotExist();*
6. *queue2.addMesssage(new CloudQueueMessage("Msg1"));*

The basic algorithm of Test Transformer is to traverse every storage item. Once a new item is visited, the Test Transformer records a pre-defined sequence of standard Azure API methods that create this item in the real storage. A transformed unit test could first call such method sequence to achieve the required cloud state, and then execute the unit under test followed by some assertions.

[1] Here, we omit some path constraints (such as constraints for field values) because they are irrelevant to the path exploration in this example.

Discussion

Correctness of Our Mock Cloud. To ensure the correctness of our mock cloud, we conduct unit testing on such cloud. For each method in our mock cloud, we write several unit tests. These unit tests can also be found in our project web site [10]. Each test passes using either the real cloud environment or our mock environment. Although our mock cloud cannot replace the local cloud emulator that provids a cloud application with an execution environment, our mock cloud indeed could simulate the basic behavior of the cloud storage.

Stateful Mock Cloud vs. Stateless Mock Cloud. By employing a stateful cloud, we make the assumption that the cloud is not modified concurrently by other processes. However, one may argue that a simplistic and stateless mock cloud is enough and any return value of a cloud API method should be valid considering that the cloud can be manipulated by other clients. In addition, a stateless mock cloud is much easier to implement. Although we should conduct thorough testing that includes all possible scenarios, in practice, developer testing mostly focuses on realistic common scenarios first.

Result of Testing *PhluffyFotos* Project. We apply our approach on one open-source project *PhluffyFotos* from *codeplex* since the code in this project frequently interacts with cloud storage services. We focus on testing the units that interact with the cloud environment. In total, we test 17 methods and our approach achieves 76.9% block coverage. Since the Azure Table Service is an extension of ADO.net data services, we also mock some of the *ADO.net* data service API methods to enable our approach to explore the methods under test. The details of the test results are shown in our project web site [10]. The results show us our approach is able to test Microsoft Azure applications with high structural coverage.

Test-Driven Development. We adopt a Test-Driven-Development-based approach to implement our mock cloud. Each time we test a new program unit, we extend our mock cloud with new functionalities used in the new unit, and then test this unit again. In general, most generated test inputs and cloud states would fail initially, and then we manually investigate the reported failures. Some failures are due to the insufficiency of the mock cloud. In these cases, we improve the mock cloud based on these reported failures. Other failures are due to the insufficiency of the parameterized unit tests such as insufficient assumptions there that could cause the generation of invalid test inputs or incorrect assertions there. Another type of failures could be due to faults in the cloud application code. However, we have not found any real fault in the already well tested application.

Conclusion and Future Work

In this article, we present an approach that combines a mock cloud and Dynamic-Symbolic-Execution to automatically test cloud applications. Currently, our approach is implemented on Pex and can be applied to Microsoft Azure applications; however, the key idea of our approach is general for any type of cloud applications adopting the Platform-as-a-Service model. Other test generation tools can be also used by our approach with different test generation techniques.

We plan to conduct more unit testing on our mock cloud and select more open source projects to evaluate our approach. In addition, we plan to extend our mock cloud with more functionalities as the real cloud environment, and extend our mock cloud to include the classes in *Microsoft. WindowsAzure.ServiceRuntime* and *Microsoft.WindowsAzure* namespaces.

References

[1] http://www.microsoft.com/windowsazure/

[2] Paul Ammann and Jeff Offutt. Introduction to Software Testing. Cambridge Univ Press, 2008.

[3] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Proceedings of ESEC/FSE-13, pages 263–272, 2005.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Proceedings of PLDI, pages 213–223, 2005.

[5] http://www.microsoft.com/windowsazure/sdk/

[6] http://www.nmock.org/

[7] http://research.microsoft.com/projects/pex/

[8] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. In Proceedings of ASE, pages 365-368, 2006.

[9] Nikolai Tillmann and Wolfram Schulte.Parameterized unit tests. In Proceedings of ESEC/FSE-13, pages 253-262, 2005

[10] https://sites.google.com/site/asergrp/projects/cloud

[11] http://phluffyfotos.codeplex.com/