Environment Modeling for 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. The behavior of a unit in a cloud application is dependent on the test inputs as well as the state of the cloud environment. Generally, manually providing various test inputs and cloud states for conducting developer testing is time-consuming and labor-intensive. To reduce the manual effort, developers could employ automated test generation tools. However, applying an automated test generation tool faces the challenge of generating various cloud states for achieving effective testing, such as achieving high structural coverage of the cloud application since these tools cannot control the cloud environment. To address this challenge, we propose an approach to (1) model the cloud environment for simulating the behavior of the real environment and, (2) apply Dynamic Symbolic Execution (DSE) to both generate test inputs and 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 cloud states to achieve high structural coverage of the cloud applications.

Keywords

Cloud Computing, Software Testing, Dynamic Symbolic Execution, Cloud Environment Model

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 installing or running 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. Unit testing provides benefits such as being able to test a unit without waiting for other units to be available, being able to detect and remove software faults at a lower cost comparing to do so at a later stage [2]. To conduct unit testing on cloud applications, a practical way is to employ various desktop-based cloud-environment emulators, such as Microsoft Azure Compute and Storage Emulators [3], which enable developers to run and test their cloud applications locally rather than testing them after deployment. In addition, developers also need to provide various test inputs. According to our empirical investigations on 21 open source projects of Azure cloud applications, 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.

Generally, testing a unit with all possible inputs is infeasible 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 the execution of any existing test case, then this fault can never be revealed. Our empirical investigation on one open source project (Lokad-Cloud) shows that 78% blocks are not covered because the existing test cases fail to provide either specific cloud states or program inputs. Since different execution paths of a unit under test require different combinations of program inputs and cloud states, developers may miss some combinations when writing test cases (including setting up a cloud state).

Manually writing test cases to achieve high structural coverage requires developers to look into the implementation details of the unit under test, making it a labor-intensive and time-consuming task. Sometimes, it is also difficult for developers to provide specific combinations of test inputs and cloud states in order to cover some specific paths or blocks. To reduce the manual effort, testers or developers could employ automated test generation tools that automatically generate test inputs to achieve high structural coverage, such as tools based on Dynamic Symbolic Execution (DSE) [4] (also called concolic testing [5]), a state-of-the-art structural testing technique.

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. When an automated test generation tool is directly applied on one open source project (PhluffyFotos [9]) in our empirical investigation, the block coverage achieved by the tool is only 6.87%.

Using a stub cloud model could alleviate those issues. With such a model, the real cloud environment can be simulated with a fake/stub one, which is capable of providing some default or user-defined return values to the cloud API method calls. However, these return values could not guarantee high structural coverage, and still require much manual effort. A parameterized cloud model extends a stub cloud model to automatically produce return values with the need of covering different paths or blocks. Since the underlying idea of any parameterized model is to assume that every return value is possible as long as it could lead to a new path, even this return value is infeasible in a real cloud environment. Without reflecting the logic or state consistency of the real cloud environment, such parameterized cloud model could cause false warnings among reported failing tests.

To address the challenge of automated test generation to achieve high structural coverage for cloud applications while causing no false warnings, in this article, we (1) propose a new approach to model the cloud environment that enables automated test generation tools to generate test inputs to achieve high structural coverage; (2) propose an automatic technique for cloud state generation, which could reflect real cloud states; (3) conduct empirical investigation and studies on open source projects of cloud applications; the results show that our approach could help improve the structural coverage achieved by an automated test generation tool.

Background

Dynamic Symbolic Execution (DSE) [4] 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 [6] is an automatic test generation tool for .NET, based on DSE. Pex has been integrated into Microsoft Visual Studio as an add-in. A key methodology that Pex supports is parameterized unit testing [7], 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 [8]). Among them, 5 projects include unit tests. We manually investigate the unit test results of the *Lokad.Cloud* project because there exist three classes that heavily interact with the cloud environment and a number of test cases are written to test almost every method in those three 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 three 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

```
public void DispatchMsg()
{
    var queueClient = this.storageAccount.CreateCloudClient();
    foreach (var queue in queueClient.ListQueues())
    {
        var msg = queue.GetMessage();
        var sucess = false;
        if (msg != null)
        {
            switch (q.Name)
            {
                case PhotoQueue:
                    //Dispatch this messgae to creat a thumbnail.
                    success = true;
                    break;
                case PhotoCleanupQueue:
                    //Dispatch this message to clean up the photo.
                    success = true;
                    break;
                default:
                    //Trace.TraceError("Unknow Queue found").
                    break;
            }
            if (success)
            {
                queue.DeleteMessage(msg);
            }
        }
    }
}
[TestMethod]
public void DispatchMsg_Test()
{
    //setup
    var storageAccount = CloudStorageAccount.DevelopmentStorageAccount;
    var queueStorageClient = storageAccount.CreateCloudQueueClient();
    var queue = queueStorageClient.GetQueueReference(PhotoQueue);
    queue.CreatIfNotExist();
    //clean message queue
    queue.Clear();
    //Prepare
    queue.AddMessage(new CloudQueueMessage("Message1"));
    //Act
    DispatchMsg(queueStorageClient);
    //Assert
    Assert.IsNull(queue.GetMessage());
}

```

Figure 1. A method under test with a unit test in the *PhluffyFotos* project [9].

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. No matter whether the test-writing developers used a white-box approach to construct the test cases or not, most blocks are not covered because they need some specific cloud states. Even when the test-writing developers adopted the same coverage criterion

as block coverage to generate the test cases; they generated these test cases manually. 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 [9]. 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.

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 non-trivial effort to construct the cloud state. Some branches of a more complex method or unit test may require some specific cloud states that cannot be easily constructed manually due to 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 these tools would fail due to the testability issue because the cloud-related API methods depend on the cloud environment that these tools cannot control.

Using stubs, the unit under test can be isolated from the cloud environment; however, it is still the responsibility of developers to simulate possible return values for each stub method. For example, developers manually provide a list of *CloudQueue* as the return value of the method *ListQueues()*. A stub method enables tools such as Pex to automatically generate various inputs and return values for the unit under test to explore different execution paths. However, such stubs generally cannot reflect the state 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 stub/fake object cannot capture this behavior, the method *GetMessage()* may return a non-null value even after the method *DeleteMessage()* has been executed. Consequently, this test case fails in the assertion at Line 46, causing a false warning.

Addressing Testing Challenge

To address the challenge of automated test generation of cloud applications, we propose a new approach to model the cloud environment and generate test inputs automatically. Given a unit of a cloud application under test, our approach consists of four parts: modeling the cloud, transforming the code under test with the Test Transformer, generating test inputs and cloud states with the Test Generator, transforming the generated unit tests with the Test Transformer.

Modeling the Cloud

A simple or naive modeling of a cloud environment generally cannot reflect the actual behavior of the real cloud environment, causing false warnings in the test results. We mainly construct a simulated cloud model and provide stub cloud API methods that replicate the effect of the corresponding API methods on the real cloud environment by performing the same operations on the cloud model. In particular, our cloud model currently focuses on providing simulated Azure storage services and 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 construct such a cloud model, 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 cloud model is constructed using an approach of Test-Driven Development (TDD), where we write stubs for different classes and functionalities incrementally driven by the demand of the unit under test rather than writing the whole stub storage services all at once. The name of each stub class starts with "stub", and ends with its original name. For example, the stub class for class *CloudQueue* is named as *StubCloudQueue* in our cloud model. 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 written stubs for all the main classes and API methods in the three storage services. Queue storage is simulated using an instance of List<*stubCloudQueue*>, where each *stubCloudQueue* is simulated using an instance of List<*stubCloudMessage*>. Blob storage is simulated using an instance of List<*stubContainer*> and each stubContainer is simulated using an instance of List<*stubIBlobItem*>. Table storage is simulated in a similar way.

Transforming the Code Under Test

With a cloud model, we execute a unit under test with the simulated environment rather than the real cloud environment. The Code Transformer redirects a unit under test to interact with our simulated 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 stubA; when a method *M* of Class *A* is invoked, this invocation is replaced by the

invocation of the simulated method in class *stubA.M*. Then the processed unit under test would now interact with our cloud model.

Generating Test Inputs and Cloud States

The Test Generator incorporates 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 containers, blobs, messages, and queues) to be inserted into the simulated 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 state 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 [1]. The algorithm for generating Queue storage states 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 slightly 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 *stubTable* 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 simpler) 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 StubCloudQueues (0 to MAX).
2.  QueueList ← Initiallize QueueStorage using an instance of List< StubCloudQueue > (N).
3.  for i from 0 to N
4.          StubCloudQueue_i ← Create a new instance of StubCloudQueue
5.          Pex chooses values for each field of StubCloudQueue_i
6.          M ← Pex choose the total number of StubCloudMessages (0 to MAX) to be inserted in to StubCloudQueue_i
7.          for j from 0 to M
8.              StubCloudMessages_ij ← Create a new instance of StubCloudMessage
9.              Pex chooses values for each field in StubCloudMessages_ij
10.             StubCloudQueue_i. MessageList.MessageAdd(StubCloudMessages_ij)
11.         end for
12.         QueueList.Add(StubCloudQueue_i)
13. end for
```

Figure 2. Algorithm for generating Queue storage states.

[1] An illustrative example can be found on our project website [8], where we explain how our approach works step by step.

Transforming Generated Unit Tests

Although our cloud model could simulate the basic behavior of the cloud storage, it cannot replace the real cloud/emulator since the real one provides a cloud application with an execution environment, and testing the code under test with only our simulated 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. In addition, for the purpose of regression testing or the requirement of third-party reviewers, we also need to provide a general format of our generated unit tests, which could set up the same cloud state before test execution. The Test Transformer transforms a generated unit test together with a generated 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 in the real cloud environment.

Discussion

Correctness of Our Cloud Model. To ensure the correctness of our cloud model, we conduct unit testing on such cloud model. For each method in our cloud model, we write several unit tests. These unit tests can also be found on our project website [8]. Each test passes using either the real cloud or our simulated cloud. Although our simulated cloud cannot replace the local cloud emulator that provides a cloud application with an execution environment, our cloud model indeed could simulate basic behaviors of the cloud storage.

Testing Results of the *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. In contrast, the block coverage achieved by Pex without using our cloud model is 6.87%. Since the Azure Table Service is an extension of ADO.net data services, we also write stub methods for 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 on our project web site [8]. The results show that our approach is able to test the *PhluffyFotos* Azure application with high structural code coverage.

Related Work. Two general techniques used for environment isolation are mocks and stubs [10]. Mocks are mainly used for behavior verification while stubs are mainly used for state verification. In this article, we primarily use our cloud model for state verification. By analyzing all uses of the real cloud API methods for substitution, stub methods can be automatically generated with behaviors [11]; however, a stub object generally cannot reflect the actual behavior of the real object, resulting in false warnings. In contrast, our cloud model could mimic the behavior of the real cloud environment. MODA [12] is an approach for automated testing of database applications. MODA provides a simulated database, which

is represented using tables. In contrast, our cloud model provides a simulated cloud environment, which is more complex than a database.

Limitations. Currently, our approach has three main limitations: (1) the capability of our generation of test inputs and cloud states is limited by the power of the constraint solver. If the constraint solver cannot solve a certain path constraint, our approach cannot generate test inputs or cloud states to cover that path; (2) our approach can work on only cloud applications that adopt the Platform-as-a-Service model; (3) our approach changes some implementations of the unit under test to behavior-equivalent ones, which are more friendly or testable for Pex to explore. This kind of changes does not modify the behavior of the unit under test. If we do not change the implementations of these API methods, we must write subs or models for them for Pex to explore.

Lessons Learned

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

Test-Driven Development (TDD). We adopt a TDD-based approach to construct our cloud model. Each time we test a new program unit, we extend our cloud model 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 cloud model. In these cases, we improve the cloud model 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 already well-tested applications under our investigation.

Conclusion and Future Work

In this article, we present an approach that combines a simulated cloud model and dynamic symbolic execution to automatically generate test inputs to achieve high structural coverage of cloud applications while causing no false warnings. Currently, our approach is implemented on Pex and can be applied to Microsoft Azure cloud applications; however, the key ideas of our approach are general for any type of cloud applications that adopt the Platform-as-a-Service model. Different cloud models can be constructed with our approach. Other test generation tools can also be used by our approach with different test generation techniques.

We plan to conduct more unit testing on our cloud model and select more open source projects to evaluate our approach. Since we find out that some API methods are not friendly or testable for Pex to

explore, we plan to compile a list of such API methods, and provide behavior-equivalent alternatives for such API methods. This list could help developers conduct unit testing with our approach. In addition, we plan to extend our cloud model with more functionalities of the real cloud environment, and extend our cloud model to include 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] http://www.microsoft.com/windowsazure/sdk/

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

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

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

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

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

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

[10] http://martinfowler.com/articles/mocksArentStubs.html

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

[12] Kunal Taneja, Yi Zhang, Tao Xie: MODA: automated test generation for database applications via mock objects. In Proceedings of ASE, pages 289-292, 2010.