Environmental Modeling for Automated Cloud Application Testing

Linghao Zhang, Xiaoxing Ma, and Jian Lu, Nanjing University

Tao Xie, North Carolina State University

Nikolai Tillmann and Peli de Halleux, Microsoft Research

// *Simulating environmental behavior and applying dynamic symbolic execution can help generate test inputs and cloud states to achieve high structural coverage for cloud applications.* //

cover image here

DEVELOPER TESTING (also called *unit testing*) can help ensure the development of high-quality cloud applications. However, unit testing normally can't take into account all possible inputs because the input space is too large or even infinite. So, developers need criteria to decide which test inputs to use and when to stop testing. Effective use of criteria such as structural-code coverage can help reveal faults.

Manually writing test cases to achieve high structural coverage requires developers to consider implementation details, making this task laborious and time-consuming. Also, developers sometimes have problems providing specific combinations of test inputs and cloud states to cover specific paths or blocks. To reduce the manual effort, they can employ automated test generation tools that use *dynamic symbolic execution* (DSE),[1] such as Pex (http://research.microsoft.com/projects/pex), a constraint-solving technique based on path exploration. But these tools fail to generate high-covering test inputs because they can't generate a required cloud state or can't control the cloud state.

Using stub cloud models could help alleviate these issues (see "Related Work" sidebar). With such models, developers can simulate a real cloud environment with a fake stub that provides default or user-defined return values to cloud-related API method calls. However, these values won't help achieve high structural coverage, and writing such a model requires much manual effort.

A parameterized cloud model extends a stub cloud model to enable generation of appropriate return values for covering different paths or blocks. Typically, a simple parameterized model would assume that every return value was possible as long as it could lead to a new path, even when this return value wasn't feasible in a real cloud environment. Without reflecting the actual cloud environment's logic or state consistency, such a model could cause false warnings among reported failing tests.

To achieve high structural coverage for cloud applications while causing few false warnings, we've developed an approach that combines a simulated cloud model and DSE. It uses

- a *test-driven-development* (TDD) process of environmental modeling to simulate the cloud environment,
- a reusable cloud model to test cloud applications, and
- an automated technique that generates test inputs and cloud states.

RELATED WORK

Testers use two general techniques for environment isolation: mocks and stubs.[1] They use mocks mainly for behavior verification and stubs mainly for state verification. Our approach primarily uses our cloud model for state verification. By analyzing all uses of actual cloud-related API methods for substitution, stub techniques can automatically generate stub methods that include behavior.[2] However, a stub object generally can't reflect an object's actual behavior, resulting in false warnings. In contrast, our cloud model can simulate the real cloud environment's behavior.

MODA (Mock Object-Based Test Generation for Database Applications) automatically tests database applications, using a simulated database represented by tables.[3] In contrast, our cloud model provides a simulated cloud environment, which is more complex than a database.

Microsoft's local cloud emulator includes complex, sophisticated code logic, posing significant challenges for path exploration. Because our cloud model is much more abstracted and simplified, DSE-based tools such as Pex can achieve more effective code coverage.

References

1. M. Fowler, "Mocks Aren't Stubs," 2 Jan. 2007; www.martinfowler.com/articles/mocksArentStubs.html.
2. N. Tillmann and W. Schulte, "Mock-Object Generation with Behavior," *Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 06), IEEE CS, 2006, pp. 365–368.
3. K. Taneja, Y. Zhang, and T. Xie, "MODA: Automated Test Generation for Database Applications via Mock Objects," *Proc. 25th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 10), IEEE CS, 2010, pp. 289–292.

We've applied this approach successfully to several open-source cloud applications running in Microsoft's Windows Azure platform (www.microsoft.com/windowsazure), which employs the *platform as a service* (PaaS) model.

Empirical Investigations

We studied 21 open-source Azure projects from CodePlex and Google Code (for a list, see our project website, https://sites.google.com/site/asergrp/projects/cloud). Five of them included unit tests. We manually investigated the results from the Lokad Cloud project because it includes three classes that heavily interact with the cloud environment and because testers have written test cases to check almost every method in them. The unit tests achieved 80 percent block coverage for the class BlobStorageProvider, 79 percent for the class QueueStorageProvider, and 93 percent for the class TableStorageProvider.

We carefully inspected the blocks that these three classes didn't cover. Of the 141 uncovered blocks, 78 percent (111 blocks) weren't covered because the existing test cases didn't provide specific cloud states or program inputs. Whether the test-writing developers used a white-box approach to construct the test cases or not, most of those blocks weren't covered because they required specific cloud states. Even when the test writers adopted the same coverage criterion as block coverage to generate the test cases, they generated the test cases manually. Different unit execution paths require different combinations of program inputs and cloud states, and developers can miss some combinations when writing test cases (including setting up cloud states).

The remaining 30 uncovered blocks weren't covered either because no test case executed the method the block belonged to or because covering the block depended on other business logic.

As our results show, generating test inputs and cloud states manually can prove challenging. In the next section, we use a real code example to illustrate similar challenges in automated testing.

The Challenge of Automated Testing

Figure 1 illustrates the testing challenge. The code snippet is a simplified method with a unit test from PhluffyFotos, an open-source project (http://phluffyfotos.codeplex.com). The method DispatchMsg first acquires a CloudQueueClient from the storageAccount at line 3 and gets a list of all existing MessageQueues at line 4. It then 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 true if the message has been successfully dispatched and processed at lines 14 and 18. Finally, the method deletes the message at line 26 if success is true.

To write test cases to check this method, developers must first clean up the cloud to ensure that the old cloud state doesn't affect the results. Then, they must prepare an appropriate cloud state before execution. An illustrative manually written test case at lines 31–47 first gets a reference of the CloudQueue PhotoQueue at line 37, cleans all the messages in the queue at line 40, inserts a new message into the queue at line 42, and then executes the method under test at line 44. The assertion at line 46 checks whether the message has been deleted.

```
1:  public void DispatchMsg()
2:  {
3:       var queueClient = this.storageAccount.CreateCloudQueueClient();
4:       foreach( var queue in queueClient.ListQueues() )
5:       {
6:            var msg = queue.GetMessage();
7:            bool success = false;
8:            if ( msg != null )
9:            {
10:                switch ( queue.Name )
11:                {
12:                     case PhotoQueue:
13:                          //Dispatch this message to create 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("Unknown 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.CreateIfNoExist();
39:      //Clean message queue
40:      queue.Clear();
41:      //Prepare
42:      queue.addMessage(new CloudQueueMessage( "Message1" ));
43:      //Act
44:      DispatchMsg();
45:      //Assert
46:      Assert.IsNull(queue.GetMessage());
47: }
```

FIGURE 1. A method under test with a unit test in the PhluffyFotos project. This figure is based on an actual code example to illustrate the challenge of automated testing.

However, if the developers want to cover all this method's branches, they must 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 example of the PhotoQueue or PhotoCleanupQueue should exist in at least one message in the queue. For this relatively simple method under test, developers must expend nontrivial effort to construct the cloud state. Covering some branches of a more complex method might require specific cloud states that can't be easily constructed manually owing to the complex execution logic.

Automated test generation tools usually must execute cloud-related API methods to collect information (by instrumenting these methods). Some tools, such as Pex, use symbolic execution to track the usage of the value a cloud-related API method returns. Depending on the subsequent branching conditions on the returned value, these tools execute the unit under test multiple times, using different values to explore new execution paths. However, directly applying these tools would fail because the return values of cloud-related API methods depend on the cloud environment's state, which these tools can't control, as we mentioned before.

Using stubs can isolate the unit under test from the cloud environment; however, the developer still must simulate possible return values for each stub method. For example, developers manually provide a list CloudQueue as the return value of the method ListQueues(). A stub method lets tools such as Pex automatically generate various inputs and stub-method return values to explore different execution paths. However, such stubs generally can't reflect state changes of the cloud environment. For example, after the method DeleteMessage(msg) at line 26 in Figure 1 executes, 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 or fake object can't capture this behavior, GetMessage() could return a non-null value even after DeleteMessage() has executed. Consequently, this test case would fail in the assertion at line 46, causing a false warning.

Addressing the Challenge

Given a unit of a cloud application under test, our approach models the cloud, transforms the code under test, generates test inputs and cloud states, and transforms the generated unit tests.

Modeling the Cloud

We construct a simulated cloud model and provide stub cloud API methods that simulate the corresponding API methods' effect on the real cloud by performing the same operations on the cloud model. Our model currently focuses on providing simulated Azure storage services and classes in the Microsoft.WindowsAzure.StorageClient namespace, which provides interactions with Windows Azure storage services. These services provide three kinds of storage:

- *Queue storage* transports messages between applications.
- *Blob (binary large object) storage* stores entities such as images, documents, and videos.
- *Table storage* provides structured storage that users can query to store collections of entities and properties.

To construct such a model, we carefully read API documents from the Microsoft Developer Network and code examples from open source projects. We use a TDD process in which we write stubs for different classes incrementally on the basis of unit demand rather than writing all the stub storage services at once. Each stub class's name starts with Stub and ends with its original name. For example, we named the stub class for the class CloudQueue in our model StubCloudQueue. Each method retains its original name.

We build up the three kinds of storage on the basis of on C# generic collections. We've written stubs for all the main classes and API methods in the three storage services. We simulate queue storage using an instance of List<StubCloudQueue>, in which we simulate each StubCloudQueue using an instance of List<StubCloudMessage>. We simulate blob storage using an instance of List<StubCloudBlobContainer> and simulate each StubCloudBlobContainer using an instance of List<StubIBlobItem>. We simulate the table storage similarly.

```
1.  N $\leftarrow$ Pex chooses the total number of StubCloudQueues (0 to MAX)
2.  QueueList $\leftarrow$ Initiallize QueueStorage using an instance of List< StubCloudQueue > (N)
3.  for i from 0 to N
4.      StubCloudQueue$_i$ $\leftarrow$ Create a new instance of StubCloudQueue
5.      Pex chooses values for each field in StubCloudQueue$_i$
6.      M $\leftarrow$ Pex chooses the total number of StubCloudMessages (0 to MAX) to be inserted into StubCloudQueue$_i$
7.      for j from 0 to M
8.          StubCloudMessages$_{ij}$ $\leftarrow$ 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. The algorithm for generating queue storage states. This algorithm describes how the entire QueueList is initialized with symbolic values.

Transforming the Code under Test

With the cloud model, we execute a unit under test with the simulated environment rather than the actual cloud environment. The *code transformer* redirects a unit under test to interact with the simulated environment. Preprocessing the unit under test makes this process possible. Specifically, if the target unit under test refers to class A from the Microsoft.WindowsAzure.StorageClient namespace, the code transformer redirects the reference to class StubA. When a method M of class A is invoked, the code transformer replaces the invocation with an invocation of the simulated method M of class StubA. Thus, the unit now interacts with our cloud model.

Generating Test Inputs and Cloud States

The *test generator* incorporates Pex to generate both test inputs and cloud states for a unit under test. Specifically, Pex generates values for not only symbolic program inputs but also symbolic cloud states. The generated values for symbolic cloud states include various storage items (such as containers, blobs, messages, and queues) to be inserted into the simulated cloud before test execution. Pex produces a final suite in which each test includes a test input and a cloud state (our project website includes an illustrative example, in which we explain how our approach works step-by-step). Figure 2 shows the algorithm for generating queue storage states.

To ensure that Pex can choose a valid value for each storage item's field, we add various constraints. 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 will choose only the name for each container, making the URI address field valid. We use a similar algorithm to generate blob storage states.

However, the algorithm to generate table storage states differs slightly. A real cloud table can store instances of TableServiceEntity from different subclasses of TableServiceEntity, but in most open source projects, each cloud table stores instances of TableServiceEntity from only a specific subclass of TableServiceEntity. So, we restrict each StubCloudTable to store instances from only one subclass of StubTableServiceEntity. (All user-defined types of table service entity are inherited from the class TableServiceEntity, which is similar to database schema but much simpler). Such simplification lets Pex easily generate table storage states without losing much applicability.

Transforming Generated Unit Tests

Although our cloud model can simulate cloud storage's basic behavior, it can't provide a cloud application with the actual cloud's execution environment. To gain high confidence on code correctness, we must test the code with either a local emulated cloud environment or the actual cloud environment. In addition, for regression testing or third-party reviewers' requirements, we must provide a general format of our generated unit tests, which could set up cloud states for the emulated or actual cloud environment before execution.

The *test transformer* transforms a generated unit test and a generated cloud state into a general unit test. Specifically, it converts a cloud state created by the test generator to a sequence of real cloud-related API methods that could construct the same state in the emulated or actual cloud environment.

Testing Our Approach

We conducted unit testing to ensure our cloud model's correctness. For each method, we wrote a number of unit tests (see our project website). Each test passes using either the real cloud or our simulated cloud. Although our simulated cloud can't replace a local cloud emulator that provides a cloud application with an execution environment, it can simulate the basic behavior of cloud storage.

We applied our approach to the PhluffyFotos project because its code frequently interacts with cloud storage services. We focused on testing the units that interact with the cloud. In total, we tested 17 methods, and our approach achieved 76.9 percent block coverage. In contrast, Pex achieved only 6.87 percent block coverage without using our cloud model.

We also applied our approach to two other open source projects. It increased block coverage from 74.3 to 100 percent on the AzureMembership project and from 50 to 91.6 percent on the disibox project. Our project website shows the details of our testing results.

Currently, our approach has two main limitations. First, the underlying constraint solver's lack of power limits the ability to generate test inputs and cloud states. If the constraint solver can't solve a certain path constraint, our approach can't generate test inputs or cloud states to cover that path. Second, our approach works only on cloud applications that adopt the PaaS model. To provide substantial empirical evidence, we must apply our approach to more cloud applications to evaluate our model and approach's validity and effectiveness.

Lessons Learned

During the development of our cloud model, we learned several lessons that might prove helpful to other developers.

Stateful vs. Stateless Cloud Model

By employing a stateful cloud model, we assume that other clients or processes haven't modified the cloud concurrently. However, some might argue that a simplistic, stateless cloud model is enough and that any return value of a cloud-related API method should prove valid, considering that other clients or processes can concurrently manipulate the cloud. Also, a stateless cloud model is much easier to construct.

We should conduct thorough testing that includes all possible scenarios. However, in practice, developer testing focuses mostly on realistic common scenarios, which a stateful cloud model represents.

TDD

After we test a new program unit, we extend our cloud model with that unit's new behavior and then retest the unit. Some generated test inputs and cloud states failed initially, so we manually investigated the reported failures. Some failures resulted from the cloud model's insufficiency. In such cases, we improved the cloud model on the basis of the failures.

Faults in the cloud application code can cause another type of failure. However, we haven't found any real faults in the well-tested applications we investigated.

Testable API Method Alternatives

We've found that developers often combine methods from other API libraries—such as ADO.NET (ActiveX Data Objects for .NET) and System.Linq—with Azure API methods (for example, methods that access the cloud state). Such methods from other API libraries aren't quite testable—that is, Pex will have trouble exploring them owing to their method implementations' high complexity. For example, a common way to make a query to a table storage service is to invoke IQueryable<T>.Where<T>(Func<TSource, bool>), which returns the query results, but this method isn't quite testable.

We've observed that for some API methods, alternative methods exist that are functionally equivalent in the context in which the unit under test used them and that might be more testable. For example, because our cloud model implements the table service using List<MockTableEntity>, one alternative way to access the table service storage is IEnumerable<T>.Where<T>(Func<TSource, bool>).ToList(), which is more testable because its implementation is simpler. Instead of spending nontrivial effort to construct a sophisticated model for these API methods, we substitute their call sites with those of more testable alternatives. Our project website lists API methods that the open source cloud application we investigated used frequently, along with alternatives.

Although we've applied our approach to Windows Azure cloud applications, it's applicable to any cloud application that adopts the PaaS model. You could also use it to construct different cloud models. In addition, our approach can employ other test generation tools and techniques.

We plan to conduct more unit testing on our cloud model and select more cloud applications for further evaluation. We also plan to continue compiling our list of untestable API methods and their behaviorally equivalent alternatives. This list could help developers improve the effectiveness of unit testing with our approach. In addition, we plan to extend our cloud model to more closely imitate the actual cloud environment, such as simulating classes in the `Microsoft.WindowsAzure.ServiceRuntime` and `Microsoft.WindowsAzure` namespaces.

Acknowledgments

US National Science Foundation grants CCF-0725190, CCF-0845272 (and its CiC Supplement), CCF-0915400, and CNS-0958235; US Army Research Office grant W911NF-08-1-0443; National Natural Science Foundation of China grants 60973044, 60736015, and 61021062; and the 973 Program of China (2009CB320702) supported this research.

Reference

1. P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. 2005 ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLID 05), ACM, 2005, pp. 213–223.

ABOUT THE AUTHORS

LINGHAO ZHANG is a PhD student in the State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology at Nanjing University. He's also a visiting scholar in the Department of Computer Science at North Carolina State University. His research interests include software testing, analysis, and verification. Zhang has a BS in computer science from Nanjing University. Contact him at lzhang25@smail.nju.edu.cn.

TAO XIE is an associate professor in the Department of Computer Science at North Carolina State University. His research interests include software engineering, particularly software testing, analysis, and analytics. Xie received a PhD in computer science from the University of Washington at Seattle. He's a member of IEEE and ACM. Contact him at xie@csc.ncsu.edu.

NIKOLAI TILLMANN is a principal research software design engineer at Microsoft Research. His research involves combining dynamic and static program analysis techniques for automatic test-case generation. Tillmann has an MS in computer science from the Technical University of Berlin. Contact him at nikolait@microsoft.com.

PELI DE HALLEUX is a senior software design engineer at Microsoft Research. His research involves combining dynamic and static program analysis techniques for automatic test-case generation and making those accessible to the masses of developers. de Halleux has a PhD in applied mathematics from the Catholic University of Louvain. Contact him at jhalleux@microsoft.com.

XIAOXING MA is a professor in the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology at Nanjing University. His current research interests include self-adaptive software systems, middleware systems, and cloud computing. Ma has a PhD in computer science from Nanjing University. He's a member of IEEE. Contact him at xxm@nju.edu.cn.

JIAN LU is a professor in the State Key Laboratory for Novel Software Technology Department of Computer Science and Technology and the director of the State Key Laboratory for Novel Software Technology at Nanjing University. His research interests include software methodologies, software automation, software agents, and middleware systems. Lu has a PhD in computer science from Nanjing University. He also serves as the director of the Software Engineering Technical Committee of the China Computer Federation. Contact him at lj@nju.edu.cn.