Environment Modeling for Automated Testing of Cloud Applications

Linghao Zhang[1,2], Tao Xie[2], Nikolai Tillmann[3], Peli de Halleux[3], Xiaoxing Ma[1], Jian Lu[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, China,
[2]North Carolina State University, USA, [3]Microsoft Research, USA

Abstract: Recently, cloud computing platforms, such as the Windows Azure platform, 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 executed test inputs as well as the state of the cloud environment. Generally, it is time-consuming and labor-intensive to manually provide various test inputs and cloud states for conducting developer testing. To reduce the manual effort, developers could employ automated test generation tools. However, applying automated test generation tools 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 generate both test inputs and cloud states to achieve high structural coverage. We apply our approach on some open-source Azure cloud applications. The results show that our approach helps generate 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 provides both virtualized hardware and software resources 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 program, 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 cloud 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 to as unit testing) could be used. Unit testing provides benefits such as testing a unit without waiting for other units to be available as well as detecting and removing software faults at a lower cost comparing to doing so at a later stage. To conduct unit testing on cloud applications, a practical way is to employ various desktop-based cloud-environment emulators, such as Windows Azure Compute and Storage Emulators, 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, some test cases are written for testing a unit that interacts with the cloud environment. Most of these test cases begin with a step of preparing the cloud-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. Effective use of coverage criteria makes it more likely that faults could be revealed. Among different coverage criteria, structural code coverage such as statement or block 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 cannot be revealed. Our empirical investigation on an open-source project (Lokad-Cloud) shows that 78% not-covered blocks are caused by the insufficiency of existing test cases failing 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 cloud states).

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 Pex [3] based on Dynamic Symbolic Execution (DSE) [2], which is a constraint-solving-based technique based on path exploration.

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 [5]) 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 cloud-related API method calls. However, these return values could not help achieve high structural coverage, and writing such a model still 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 is possible as long as it could lead to a new path, even when this return value is infeasible in a real cloud environment. Without reflecting the logic or state consistency of the real cloud environment, such a 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 few false warnings, in this article, we present an approach that combines a simulated cloud model and DSE to automatically generate test inputs. This article makes the following main contributions: (1) a Test-Driven-Development (TDD) based process of environment modeling for simulating the cloud environment for cloud applications; (2) a cloud model, which is available and reusable for others to test their Azure cloud applications; (3) an automated test generation technique that generates both test inputs and cloud states to achieve high structural coverage while causing few false warnings; (4) empirical investigations on applying the approach on open-source projects of cloud applications; (5) identification of a list of frequently used cloud-related API methods that pose testability issues for a DSE-based test generation tool.

Empirical Investigations

We collect open-source Azure projects available from *Codeplex* and *Google Code* (21 projects in total, whose details can be found on our project website [4]). 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 written to test almost every method in these 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 for causing a block not to be 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 total, there exist 141 not-covered blocks, 78% (111/141) of which are not covered because of the insufficiency of the existing test cases 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 of these 111 not-covered 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 the developers may miss some combinations when writing test cases (including setting up cloud states).

```
public void DispatchMsg()
{
    var queueClient = this.storageAccount.CreateCloudClient();
    foreach( var queue in queueClient.ListQueues() )
    {
        var msg = queue.GetMessage();
        bool success = false;
        if ( msg != null )
        {
            switch ( queue.Name )
            {
                case PhotoQueue:
                    //Dispatch this message to create a thumbnail.
                    success = true;
                    break;
                case PhotoCleanupQueue:
                    //Dispatch this message to clean up the photo.
                    success = true;
                    break;
                default:
                    //Trace.TraceError("Unknown  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.CreateIfNoExist();
    //Clean message queue
    queue.Clear();
    //Prepare
    queue.addMessage( new CloudQueueMessage( "Message1" ));
    //Act
    DispatchMsg();
    //Assert
    Assert.IsNull(queue.GetMessage());
}
```

Figure 1. A method under test with a unit test in the *PhluffyFotos* project [5].

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* [5]. The method *DispatchMsg* first acquires a *CloudQueueClient* from the *StorageAccount* at Line 3 and gets a list of all 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 18. 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 results, 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 the method under test 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 relatively simple method under test, developers already need some non-trivial effort to construct the cloud state. Covering some branches of a more complex method may require some specific cloud states that cannot be easily constructed manually due to the complex execution logic.

Automated test generation tools usually require executing cloud-related API methods (by instrumenting these methods) to collect necessary information for test generation. Particularly, some tools, such as Pex, use symbolic execution to track the usage of the value returned by a cloud-related API method. 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 where the return values of 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 stub-method return values for the unit under test to explore different execution paths. However, such stubs generally cannot reflect 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()* could return a non-null value even after the method *DeleteMessage()* has been executed. Consequently, this test case would fail 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 Code 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 the cloud environment generally cannot reflect the actual behavior of the real cloud environment, causing false warnings in test results. We mainly construct a simulated cloud model and provide stub cloud API methods that simulate the effect of the corresponding API methods on the real cloud 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 Windows Azure storage services. Windows Azure storage services provide three kinds of storage: *Blob* (abbreviation for Binary Large Object), which is used to store entities such as images, documents, and videos; *Table*, which provides queryable structured storage to store 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 API documents from MSDN but also read through many code examples from the investigated open-source projects. We construct our cloud model using a process of Test-Driven Development (TDD), where we write stubs for different classes 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. The Queue storage is simulated using an instance of *List*<*stubCloudQueue*>, where each *stubCloudQueue* is simulated using an instance of *List*<*stubCloudMessage*>. The Blob storage is simulated using an instance of List<*stubContainer*> and each *stubContainer* is simulated using an instance of List<*stubIBlobItem*>. The 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* from 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 an invocation of the simulated method *M* of class *stubA*. 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 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 the execution of the unit under test. 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 Blob storage states. But the algorithm to generate Table storage states is slightly different. Practically, different types of table entities can be stored in the same cloud table, but most open-source projects use one cloud table to store only a particular type of table entities. Therefore, we also restrict each *stubTable* to store only one type of entities. The algorithm for generating Table storage states 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 to more easily 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$_N$ ← Create a new instance of StubCloudQueue
5.           Pex chooses values for each field in StubCloudQueue$_N$
6.           M ← Pex chooses the total number of StubCloudMessages (0 to MAX) to be inserted into StubCloudQueue$_N$
7.           for j from 0 to M
8.                 StubCloudMessages$_{NM}$ ← Create a new instance of StubCloudMessage
9.                 Pex chooses values for each field in StubCloudMessages$_{NM}$
10.                StubCloudQueue$_{MN}$. MessageList.MessageAdd(StubCloudMessages$_{NM}$)
11.          end for
12.          QueueList.Add(StubCloudQueue$_N$)
13.  end for
```

Figure 2. Algorithm for generating Queue storage states.

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, it is necessary to test the code with either the local emulated cloud environment or the real cloud environment. In addition, for regression testing or requirements of third-party reviewers, we also need to provide a general format of our generated unit tests, which could

[1] An illustrative example can be found on our project website [4], where we explain how our approach works step by step.

set up cloud states for the emulated or real cloud environment 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-related API methods that could construct the same state in the emulated or 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 a number of unit tests. These unit tests can also be found on our project website [4]. Each test passes using either the real cloud or our simulated cloud. Although our simulated cloud cannot replace a local cloud emulator, which provides a cloud application with an execution environment, our cloud model indeed could simulate the basic behavior of the cloud storage.

Testing Results. We apply our approach on one open-source project *PhluffyFotos* [5] 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 only 6.87%. We also apply our approach on two other open-source projects. Our approach helps increase the block coverage from 74.3% to 100.0% for the *AzureMembership* project, and from 50.0% to 91.6% for the *disibox* project. The details of the testing results are shown on our project website [4]. The results show that our approach is able to test these applications with high structural code coverage.

Related Work. Two general techniques used for environment isolation are mocks and stubs [6]. Mocks are mainly used for behavior verification while stubs are used for state verification. In this article, we primarily use our cloud model for state verification. By analyzing all uses of the real cloud-related API methods for substitution, stub methods can be automatically generated with behavior [7]; however, a stub object generally cannot reflect the actual behavior of the real object, resulting in false warnings. In contrast, our cloud model could simulate the behavior of the real cloud environment. MODA [8] 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. In addition, we also find out that some cloud-related API methods could be replaced by some behaviorally-equivalent ones, which enable test generation tools to perform better. Compared to the local cloud emulator provided by Microsoft, our cloud model is much more abstracted and simplified so that DSE-based tools such as Pex could achieve satisfactory effectiveness in code coverage, since the local cloud emulator includes very complex and sophisticated code logic, posing significant challenges for path exploration.

Limitations. Currently, our approach has three main limitations. First, the capability of generating test inputs and cloud states is limited by the power of the underlying 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. Second, our approach can work on only cloud applications that adopt the Platform-as-a-Service model. Third, to provide substantial empirical evidence, we need to apply our approach to more cloud applications to evaluate the validity and effectiveness of our model and approach.

Lessons Learned

Stateful Cloud Model vs. Stateless Cloud Model. By employing a stateful cloud model in our approach, 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-related API method should be valid considering that the cloud can be concurrently 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, as focused by a stateful cloud model.

Test-Driven Development (TDD). We adopt a TDD-based process to construct our cloud model. Each time when we test a new program unit, we extend our cloud model with new behavior used in the new unit, and then test this unit again. In general, some 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. 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.

Testable API Method Alternatives. During the construction of our cloud model, we find that some API methods from other API libraries (such as *ADO.NET* and *System.Linq*) are often used together with Azure API methods, such as methods that access the cloud state. These API methods from other API libraries are not quite testable, i.e., posing testability issues for Pex to explore them, due to high complexity of their method implementations. For example, to make a query to the table storage service, a commonly used way is to invoke *IQueryable<T>.Where<T>(Func<TSource, bool>)*, which would return the result of a query, but this API method is not quite testable. We observe that some of these API methods have alternative functionally-equivalent ones in the context of their usage by the unit under test, and these alternative API methods may be more testable. Considering that the table service is implemented using *List<MockTableEntity>* in our cloud model, we could use an alternative way to access the table service storage, such as *IEnumerable<T>.Where<T>(Func<TSource, bool>).ToList()*, which is more testable due to its simpler implementation. In particular, instead of spending non-trivial effort on constructing a sophisticated model for these API methods, we substitute the call sites of these API methods with call sites of their more testable alternative ones. Such API methods that are frequently used in the investigated open-source cloud application (together with their alternative ones) are listed on our project website [4].

Conclusion

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 few false warnings. Currently, our approach is implemented upon Pex and can be applied to Windows Azure cloud applications; however, the key ideas of our approach are general for other types of cloud applications that adopt the Platform-as-a-Service model. Different cloud models can be constructed following 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 cloud applications to evaluate our approach. Since we find out that some API methods are not testable for Pex to explore, we plan to compile a list of such API methods, and provide behaviorally-equivalent alternatives for such API methods. This list could help developers improve effectiveness of unit testing with our approach. In addition, we plan to extend our cloud model to simulate more parts of the real cloud environment, such as simulating classes in the *Microsoft.WindowsAzure.ServiceRuntime* and *Microsoft.WindowsAzure* namespaces.

Acknowledgment. This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1- 0443, NSFC (60973044, 60736015, 61021062), and the 973 Program of China (2009CB320702).

References

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

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

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

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

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

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

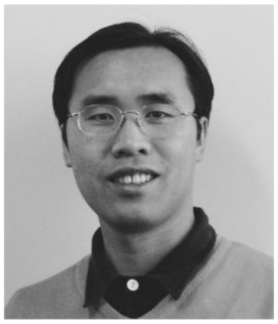
[7] Nikolai Tillmann and Wolfram Schulte. Mock-object generation with behavior. In Proceedings of ASE, pages 365-368, 2006.

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

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 is 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 received his BS in computer science from Nanjing University. Contact him at lzhang25@smail.nju.edu.cn.

Mailing Address: State Key Laboratory for Novel Software Technology, Nanjing University,
Room 812, Comp. Sci. & Tech. Building,
Xianlin Campus, 163 Xianlin Avenue, Qixia, Nanjing 210046, Jiangsu, China.
Email: 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 is a member of IEEE and ACM. Contact him at xie@csc.ncsu.edu.

Mailing Address: Department of Computer Science, North Carolina State University,
Campus Box 8206, Raleigh, NC 27695-8206.
Phone: +1 919 515-3772
Fax: +1 919 515-7896
Email: 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 received his MS in computer science from the Technical University of Berlin. Contact him at nikolait@microsoft.com.

Mailing Address: Microsoft Research,
One Microsoft Way,
Redmond, USA.
Phone: +1 425 707-6031
Fax: +1 425 936-7329

Email: 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 received his PhD in applied mathematics from the Catholic University of Louvain. Contact him at jhalleux@microsoft.com.

Mailing Address: Microsoft Research,
One Microsoft Way,
Redmond, USA.
Phone: +1 (425) 703-6917
Fax: +1 425 936-7329
Email: jhalleux@microsoft.com

Xiaoxing Ma is a professor in the Department of Computer Science and Technology at Nanjing University. His current research interests are self-adaptive software systems, middleware systems, and cloud computing. Ma received his PhD in Computer Science from Nanjing University. He is a member of IEEE. Contact him at xxm@nju.edu.cn.

Mailing Address: State Key Laboratory for Novel Software Technology, Nanjing University,
Room 816, Comp. Sci. & Tech. Building,
Nanjing University, Xianlin Campus,
163 Xianlin Avenue, Qixia, Nanjing 210046, Jiangsu, China.
Telephone: +86 25 89686068
Fax: +86 25 83593283
Email: xxm@nju.edu.cn.

Jian Lu is a professor in the 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 received his BS, MS, and 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.

Mailing Address: State Key Laboratory for Novel Software Technology, Nanjing University,
Room 820, Comp. Sci. & Tech. Building,
Nanjing University, Xianlin Campus,
163 Xianlin Avenue, Qixia, Nanjing 210046, Jiangsu, China.
Telephone: +86 25 83593283
Email: lj@nju.edu.cn