A Model-Based Framework For Cloud API Testing

Junyi Wang , Xiaoying Bai , Linyi Li , Zhicheng Ji , Haoran Ma
Department of Computer Science and Technology
Tsinghua University
Beijing, China
baixy@tsinghua.edu.cn, wangjuny15@mails.tsinghua.edu.cn

***Abstract*—Following the Service-Oriented Architecture, a large number of diversified Cloud services are exposed as Web APIs (Application Program Interface), which serve as the contracts between the service providers and service consumers. Due to their massive and broad applications, any flaw in the cloud APIs may lead to serious consequences. API testing is thus necessary to ensure the availability, reliability, and stability of cloud services. The research proposes a model-based approach to automating API testing. The semi-structured API specifications, like HTML specifications, are gathered from the Web sites using web crawlers, and translated into XML/YAML-encoded standard representations. A scenario editor is designed to specify the dependencies among API operations. Test generators are built to derive test scripts from the specifications and scenarios, including test data, test cases for individual operations as well as operations sequences. Various algorithms can be used for test generation, such as combinatorial data generation, heuristic graph search, and optimization algorithms. The produced test scripts, together with a load model, can be deployed on Cloud and scheduled for execution. A prototype system, called ATCloud, was constructed to illustrate the process of API understanding, test scenario modeling using directed diagraph annotated with transfer probabilities between operations, cloud-based test resources management, distributed workload simulation, and performance monitoring.**

***Keywords*—API testing; model-based testing; test automation; cloud computing**

I. Introduction

With the development of SaaS (Software-as-a-Service), the concept of API (Application Program Interface) has been extended beyond interfaces of program libraries [1]. It has become the standard way for encapsulating software functions as decoupled services. Services exposed by standard APIs shield implementation details and heterogeneity, enabling dynamic binding, invocation and composition of services. With well-defined interfaces, it can effectively support software reuse, interoperability and scalability at various granularities. APIs, especially Web APIs for SaaS, are the contracts between service providers and service consumers. On one hand, APIs are used to declare commitments of functionalities by service providers. On the other hand, APIs are the standards that need to be followed strictly when programming and invocation for service consumers. Following the Service-Oriented Architecture, APIs get increasing popularization. A large number of large enterprises such as banks (e.g. mobile apps), airlines (e.g. real-time flight updates), or government (e.g. eGov, open data) have been using APIs to provide services such as data queries and functionalities, such that online services can be composed dynamically and flexibly into various business processes to adapt to agile consumer expectations. According to Programmable Web report [3], there have been over 15,000 APIs available nowadays, a considerable increase from around 200 APIs in 2005.

However, the inherent open, collaborative, and dynamic characteristics of Web APIs raise new threats to the quality of systems [2]. As APIs are exposed for open access by a large number of users on the Internet with the development of SaaS, a defect in an open API may be wide spread, causing software failures in a large scale. API testing is thus becoming necessary to ensure the quality and reliability of APIs for individual services and composite services. Cloud platforms, such as Amazon, Azure, and Ali, provide APIs for various services including computing services, storage services, data services, infrastructure services, security services, and more and more rich application services. As Cloud APIs are widely used and continuously evolve online, it generates a pressing need of an automatic testing approach to constantly detecting the changes and potential defects in the services [4]. Particularly, it poses following requirements for Cloud API testing.

- The number of APIs is large. It needs a lot of resources to generate, manage, and execute test cases. Testing is expensive, thus it is necessary to optimize test cases and test resources.
- APIs are open to various usage scenarios. Many defects are showed up in complicated situations. Intelligent techniques are needed to generate sophisticated scenario testing, such as complex operation sequences, abnormal conditions, and so on.
- Cloud is promising to achieve economies of scale. Massive scalability testing with various workloads is thus critical for Cloud performance analysis.

In counter to these needs, the paper reported ATCloud, a model-based automatic testing framework and a prototype system to support Cloud API testing. As shown in Fig. 1, ATCloud composes five main modules: Test Resource Manager, API Understanding, Test Generation, Test Engine and Test Analyzer.

1) Test Resource Manager. It manages all the test resources of the users, allocates the resources for the test and provides a web-based interface, where users can view test resources, define test tasks, build test environment, set test parameters, and track test results. Administrators can monitor test resources and test tasks.

2) API Understanding. It automatically gathers the API specifications from Cloud websites, interprets the syntax and semantics of service data and operations, and transforms it into internal semi-formal representations.

3) Test Generation. With built-in strategies, test cases are generated at different levels using different algorithms.

4) Test Engine. Test cases are encoded in executable scripts to be deployed at host environment, triggered on schedule. In this research, a test Cloud was built to dynamically allocate testing resources on demand across platforms.

5) Test Analyzer. Test results are collected, verified and validated against requirements. A monitor was built to gather performance indicators, visualize and report system status during testing.

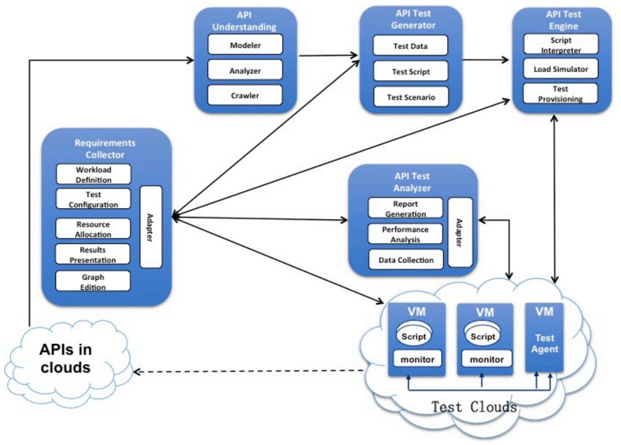


Fig. 1. Approach Overview

The rest of this paper is organized as follows. Section II presents the design and key techniques of the proposed framework, including API Understanding, the method and algorithms of test generation, massive scalable workload simulation, and the monitoring mechanism of Test Analyzer. Section III introduces a prototype system of the framework. Section IV reviews related work. Section V finally concludes this paper.

II. The Model-Based Testing Framework

A. API Understanding

APIs are usually described in natural language, including function signatures and descriptions, which contain parameter data type, return value data type and the constraints and conditions of function invocation. Taking the elastic computing services as example, the function for stopping a virtual machine instance must be invocated after the function for starting the instance. To generate tests automatically from such API definition, it needs to transform API documents into machine-interpretable standard specifications. Fig. 2 shows the process of API Understanding, which crawls Cloud services, interprets API specifications, and transforms them into YAML/XML-encoded description.

Fig.3 shows an example of XML-encoded API specification, which could be further interpreted by test generator for producing test scripts. The standard specification models an API from three aspects: data modeling, individual API modeling and API Scenario Modeling.

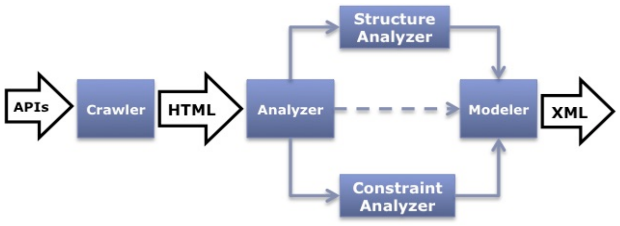


Fig. 2. API Understanding

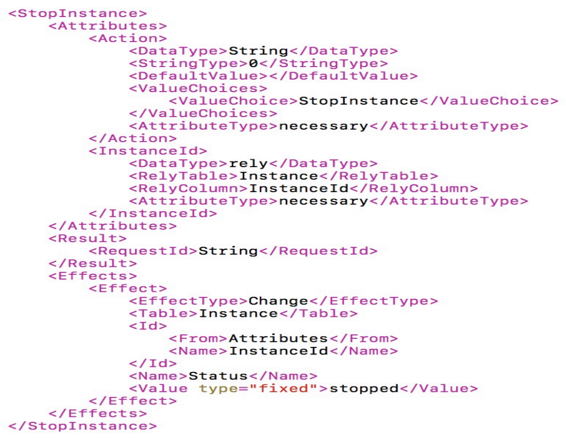

```
<StopInstance>
    <Attributes>
        <Action>
            <DataType>String</DataType>
            <StringType>0</StringType>
            <DefaultValue></DefaultValue>
            <ValueChoices>
                <ValueChoice>StopInstance</ValueChoice>
            </ValueChoices>
            <AttributeType>necessary</AttributeType>
        </Action>
        <InstanceId>
            <DataType>rely</DataType>
            <RelyTable>Instance</RelyTable>
            <RelyColumn>InstanceId</RelyColumn>
            <AttributeType>necessary</AttributeType>
        </InstanceId>
    </Attributes>
    <Result>
        <RequestId>String</RequestId>
    </Result>
    <Effects>
        <Effect>
            <EffectType>Change</EffectType>
            <Table>Instance</Table>
            <Id>
                <From>Attributes</From>
                <Name>InstanceId</Name>
            </Id>
            <Name>Status</Name>
            <Value type="fixed">stopped</Value>
        </Effect>
    </Effects>
</StopInstance>
```

Fig. 3. Example API XML Specificaiton

1) *Domain Knowledge Capture*

It captures domain knowledge such as data, operation definition and some other constraints and conditions for Cloud APIs. As showed in Fig. 2, API Crawler crawls the cloud service's web site for the descriptions of API interfaces published in HTML format on the Cloud Platform. Using the key concept extraction method, API Analyzer analyzes the captured HTML files, including structural analysis and condition analysis, in which structural analysis is mainly to extract data and operational definitions, such as function parameters, return values and names of function.

A key issue in API understanding is the modeling of domain knowledge in terms of constraints and conditions. For example, a constraint of parameter *InstanceName* says that: *"the name of the instance is required to contain [2,128] English or Chinese characters, and it must be in uppercase letters or "-". InstanceId, which default is instance, cannot start with 'http: //' or 'https: //'"*. This type of constraint is critical to system robustness and reliability testing, but usually insufficiently defined in interface specification. To enhance the modeling capability with interface semantics, it has a great potential to improve test effectiveness and efficiency. However, it is usually hard to obtain such constraints. In case APIs are well-defined and documented, some constraints could be extracted using natural language processing techniques. In many cases, it requires additional manual efforts to feed in the formalized specifications.

2) Data Modeling

In order to precisely capture data semantics and to enhance the intelligence of test data generation, the data types of service parameters are modeled with properties and constraints

ATCloud defines a data model from three perspectives: basic data types, domain concepts, and properties and constraints. Basic data types are often the meta-data of domain objects, such as Integer, String, Boolean, and so on. Domain concepts are domain objects with constraints, or user-defined composite objects, such as MAC type, IP type, and TIME. Properties and constraints are identified to model concept semantics. For example, the constraints for *CreateTopic* says that: *"The subject name is required to be a string of no more than 256 characters, must be preceded by a letter or number, and the remainder can contain letters, numbers, and crossed lines (-)."*

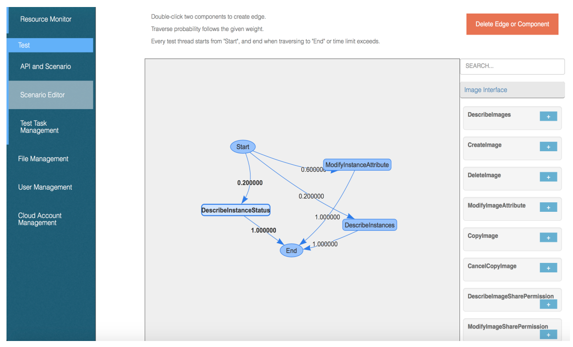
3) Individual API Modeling

This is to characterize the functionalities provided by an individual API service, which is defined by a 6-tuple as follows:

Service: =<ID, Name, Input, Output, Return, Constraint>

ID is the unique identifier for individual API service to be tested. *Name* is its name. *Input/Output* is the set of input/output parameters. *Return* represents the return value. *Constraint* represents the constraints and conditions to invoke the API service.

4) API Scenario Modeling



Double-click two components to create edge.
Traverse probability follows the given weight.
Every test thread starts from "Start", and end when traversing to "End" or time limit exceeds.

Fig. 4. An Example API Scenario Model.

A complex scenario by composing multiple APIs executed in sequence usually has a higher potency to reveal defects than single API testing [6]. Based on models for individual APIs, API Scenario Modeler builds scenario models for complex usage scenarios with multiple APIs. To capture API composition scenarios, a Directed Transfer Graph is defined to model the control flow and data flow among API invocations, as shown in Fig. 4. A node in the graph represents an API operation, and a directed link between two nodes represents a valid execution sequence between two operations. The link can be a conditional transfer, which is annotated with a Boolean expression of the conditions, or an iteration, which is annotated with the number of repeated times. In addition, the link is annotated with transfer probability to identify the differences in invocation frequencies among different operations.

Using the Transfer Matrix to represent the Directed Transfer Graph, a formal definition of the scenario model is defined as follows:

Scenario: =< ID, Name, Description, Services, Matrix, Start, End >

ID is the unique identifier for API scenario to be tested. *Name* is its name. *Description* describes the main functions of the scenario. *Services* represent the API services participating in the scenario. *Matrix,* which is generated after traversing the directed transfer probability diagraph, records the transfer probability between any two services in the graph. *Start /End* identifies the entrance and exit service of the scenario.

5) Workload modleing

The workload model is defined by a 5-tuple as follows:

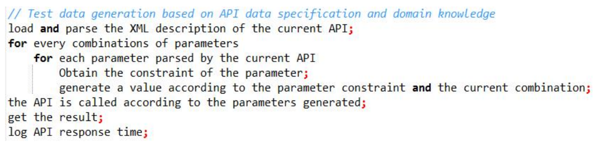
Workload: =< Duration, Size, ThinkTime, WeightGroup, ConfGroup>

Duration defines the time duration from the first request to the last request sent of all users. After the end of the duration, un-finished tasks will be interrupted. *Size* represents the number of simulated users. *ThinkTime* represents the time interval that each user follows when submitting a task request. *WeightGroup* is a vector, where each element represents the weight of the corresponding user group, and the sum of all the elements in the vector is 1. *Size* and *WeightGroup* together determine the number of users per user group. *ConfGroup* is also a vector, where each element represents the task assignment in the task request for each user in the corresponding user group. The parameters of user groups are set following certain statistical distribution models such as significant constant distribution and Markov chain model.

B. API Test Generation

Based on API models, which includes data models, individual API models and API scenario models, ATCloud generates test cases at four perspectives: (1) Test data generation based on API data specification and domain knowledge. (2) Test operation generation based on individual API modeling. (3) Operation sequence generation by traversing the paths in the directed graph of the scenario model. (4) Workload generation by generating the operation sequences following the statistical distributions of the transferring probabilities in the scenario model.

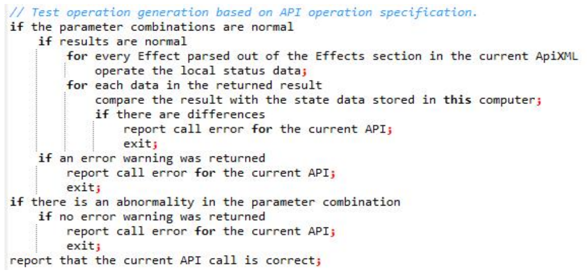
1) Test data generation based on API data specification and domain knowledge. As data modeling of ATCloud models domain data with their associated constraints, individual API modeling then models the parameters of interface functions using the pre-defined data types. Data values can be participated into equivalent classes of valid (invalid) data inputs for triggering normal (abnormal) service executions. API Test Generator then analyzes the data model and data constraints, using data partitioning, pattern matching, boundary values, constraint solving, and multi-parameter combination, to generate some normal and abnormal values of the API interface invocating parameters. Fig. 5 sketches the algorithm outline of test data generation.



```
// Test data generation based on API data specification and domain knowledge
load and parse the XML description of the current API;
for every combinations of parameters
    for each parameter parsed by the current API
        Obtain the constraint of the parameter;
        generate a value according to the parameter constraint and the current combination;
the API is called according to the parameters generated;
get the result;
log API response time;
```

Fig. 5. Test data generation algorithm

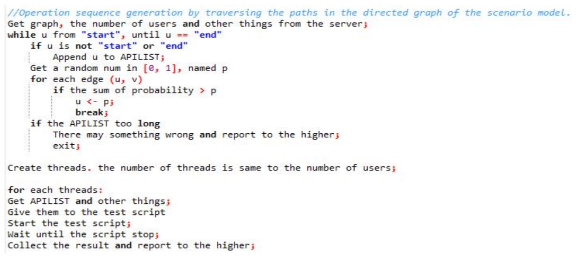
2) Test operation generation. API Test Generation module uses combinatorial testing strategies, and decision table techniques to generate test operations for each individual API service. In order to trigger normal or abnormal API executions, the data values of each parameter can be participated into equivalent classes of valid or invalid data inputs. Combinatorial testing strategies are used to select data partitions of each API parameter, as to cover various executions of a service. As a simple and intuitive way to list all possible solutions for complex combination problems, decision tables are constructed to specify the expected correspondences among inputs, outputs and return value. By dividing data partitions for input parameters of API automatically, ATCloud can achieve data coverage. It adopts combinatorial algorithms to create a decision table for API, which uses data partitions of input parameters as condition attributes and return parameter as decision attribute. REST protocol is used to invocate scripts, in order to obtain the test results. It selects a specific set of test data, invocate the API interface function to be tested, verify the return value, and record the response time. The algorithm is described as Fig. 6.



```
// Test operation generation based on API operation specification.
if the parameter combinations are normal
    if results are normal
        for every Effect parsed out of the Effects section in the current ApiXML
            operate the local status data;
        for each data in the returned result
            compare the result with the state data stored in this computer;
            if there are differences
                report call error for the current API;
                exit;
    if an error warning was returned
        report call error for the current API;
        exit;
if there is an abnormality in the parameter combination
    if no error warning was returned
        report call error for the current API;
        exit;
report that the current API call is correct;
```

Fig. 6. Test operation generation algorithm

3) Operation sequence generation. By traversing the paths in the Directed Transition Diagraph of the scenario model, test cases for operation sequences are generated, which are in the form of API sequences with the transition conditions, such as the result of a Boolean expression of the conditions, an iteration which is annotated with the number of repeated times, or transfer probability which can identify the differences in invoices among different operations. The algorithm is described as Fig.7.



```
//Operation sequence generation by traversing the paths in the directed graph of the scenario model.
Get graph, the number of users and other things from the server;
while u from "start", until u == "end"
    if u is not "start" or "end"
        Append u to APILIST;
    Get a random num in [0, 1], named p
    for each edge (u, v)
        if the sum of probability > p
            u <- p;
            break;
    if the APILIST too long
        There may something wrong and report to the higher;
        exit;

Create threads. the number of threads is same to the number of users;

for each threads:
Get APILIST and other things;
Give them to the test script
Start the test script;
Wait until the script stop;
Collect the result and report to the higher;
```

Fig. 7. Operation sequence generation

C. *Massive Scalable Workload Simulation*

On-demand resource allocation, high scalability and elasticity are the critical promises of cloud platforms [4]. To validate the performance of these promises, it needs to generate large-scale test load by simulating the massive invocations of Cloud APIs following various workload models. Workloads can be derived by generating the operation sequences following the statistical distributions of the transferring probabilities in the scenario model. The workload model, represented by the transfer probabilities, can also be obtained by analyzing execution logs of the Cloud services to learn and characterize a variety of application traces.

Various Cloud platforms exist worldwide, such as Amazon, Azure, and Ali. The datacenters of each cloud platform are distributed located. The distance between cloud servers and clients can influence the performance of cloud platforms. To simulate large-scale API invocations from geographically dispersed clients, ATCloud maintains a pool of Cloud resources from various vendors and data centers to provision test resources on demand, test tasks are wrapped, scheduled and remotely deployed to available VM(Virtual Machine) instances[7]. A distributed concurrent programming language named Erlang [8] is used to simulate the load generation, as well.

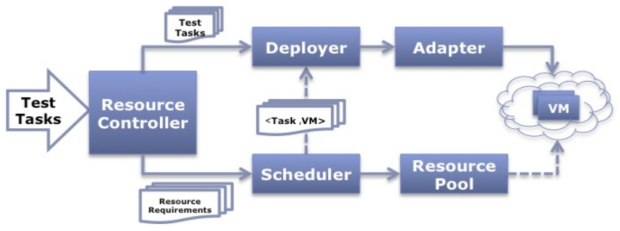


Fig. 8. On-demand resource allocation.

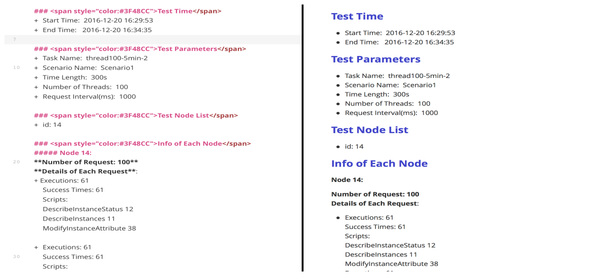
As shown in Fig. 8, ATCloud maintains a test resource pool of virtual machines across multiple Cloud platforms. After parsing test requirement, the API Test Engine will create several test tasks accordingly. Then these test tasks will be passed to Resource Manager. The Resource Manager can manage the resource pool and allocate test resources on demand during the test. The working process is as follows:

- Resource Controller extracts resource requirements such as the IaaS platform [9], compute capacity requirements from these test tasks and submit them to the Scheduler. In the meantime, these test tasks will be sent to Deployer waiting for resources to deploy.
- Scheduler maintains a Resource Pool, which contains the records of registered resources. After receiving resource requirements, Scheduler will search Resource Pool to find the most appropriate instances for each test task. In the absence of sufficient resources, Scheduler will also apply to the IaaS provider for new resources [7].
- The schedule result will be returned to Deployer, which manages virtual machines (such as creating, starting,

shutting down, cloning, and deleting virtual machines), through adapters provided by the cloud platforms. The adapter translates these operations into platform-specific orders for cloud platforms to implement.

D. *Test Analyzer*

Test Analyzer collects test results, verified and validated against requirements. Test reports, which are in markdown or html format (as showed in Fig. 9) can be downloaded for further use.



```
### <span style="color:#3F48CC">Test Time</span>
+ Start Time:  2016-12-20 16:29:53
+ End Time:   2016-12-20 16:34:35

### <span style="color:#3F48CC">Test Parameters</span>
+ Task Name:  thread100-5min-2
+ Scenario Name:  Scenario1
+ Time Length:  300s
+ Number of Threads:  100
+ Request Interval(ms):  1000

### <span style="color:#3F48CC">Test Node List</span>
+ id: 14

### <span style="color:#3F48CC">Info of Each Node</span>
##### Node 14:
**Number of Request: 100**
**Details of Each Request**:
+ Executions: 61
   Success Times: 61
   Scripts:
   DescribeInstanceStatus 12
   DescribeInstances 11
   ModifyInstanceAttribute 38

+ Executions: 61
   Success Times: 61
   Scripts:
```

Test Time

- Start Time: 2016-12-20 16:29:53
- End Time: 2016-12-20 16:34:35

Test Parameters

- Task Name: thread100-5min-2
- Scenario Name: Scenario1
- Time Length: 300s
- Number of Threads: 100
- Request Interval(ms): 1000

Test Node List

- id: 14

Info of Each Node

Node 14:

Number of Request: 100
Details of Each Request:

- Executions: 61
  Success Times: 61
  Scripts:
  DescribeInstanceStatus 12
  DescribeInstances 11
  ModifyInstanceAttribute 38

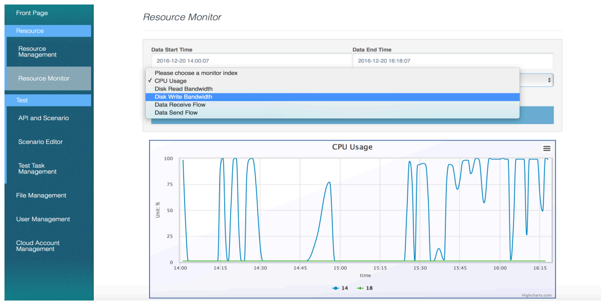
Fig. 9.   Test reports

Test Analyzer also builds a monitor, which can gather performance indicators, visualize and report system status during testing. It tracks software execution, responds to policy violations, verifies service reliability attributes, observes system behavior, generate performance reports, and plot performance graphs, as shown in Fig.10.

As different cloud platforms usually define their individual performance indicators and provide limited access using specific API, ATCloud builds integrated cloud performance model and adapter architecture to match unified SLA (Service Level Agreement) queries to various data collection interfaces. An adapter is implemented for each cloud platform to collect performance data by open APIs. The performance data are wrapped using the unified performance model as follow:

Performance: =<CloudName, Resource, Indicator, Statistics, Unit >

CloudName is the name of the cloud platform, which is being monitored. *Resource* is resource types, such as CPU, memory, Data Send Flow, and so on. *Indicator* describes property of the recourse, such as Usage, Write, Read, and so on. *Statistics* is the performance data aggregation method, for example Average, Maximum and Minimum. *Unit* represents performance's unit of measure.



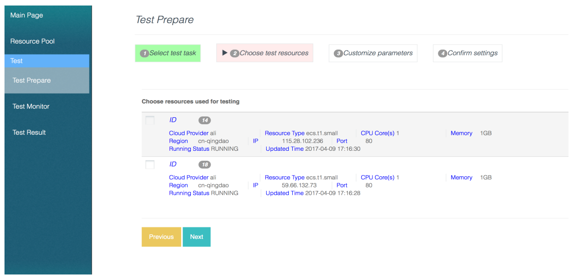
Data Start Time 2016-12-20 14:00:07
Data End Time 2016-12-20 16:18:07

Fig. 10. Performance graphs

III. A PROTOTYPE SYSTEM

A prototype system is implemented to illustrate the proposed framework. It is composed of a front-end server, a controller server, a performance monitor server, and a central database server. Two private clouds Ali and Azure are deployed. Each cloud is composed of two physical servers.

The web UI front-end is developed for the whole system using Bootstrap. Testers can choose API or API scenario for testing, configure test resources, and schedule test executions, as shown in Fig.11. Administrator testers can generate test scenarios from the front-end graphical interface. As shown in Fig.4, the node, which represents individual API can be created by clicking the plus sign above the box on the right side of the toolbar. The lines, which connect nodes and indicate the flow of the control and the data can be created by clicking the two nodes. The system limits the direction of the connections, according to actual meaning.



Choose resources used for testing

ID 14
Cloud Provider ali | Resource Type ecs.t1.small | CPU Core(s) 1 | Memory 1GB
Region cn-qingdao | IP 115.28.102.236 | Port 80
Running Status RUNNING | Updated Time 2017-04-09 17:16:30

ID 18
Cloud Provider ali | Resource Type ecs.t1.small | CPU Core(s) 1 | Memory 1GB
Region cn-qingdao | IP 59.66.132.73 | Port 80
Running Status RUNNING | Updated Time 2017-04-09 17:16:28

Fig. 11. test generation defination

During execution, the performance monitor can collect performance data and monitor the execution state and resource utilization (as showed by figure 10). During test, Web UI front-end provides the interface for testers to choose the performance indicator which to be monitored. After test, it provides the functionality of test result and the selected performance data for browsing and downloading.

IV. RELATED WORK

Model-based testing derives concrete test cases from an abstract test model, which describes entire sets of possible tests. Test models are usually designed based on the specification [10]. Various modeling techniques have been proposed for depicting software behavior, such as Extended Finite State Machine (EFSM), Specification Description Language (SDL), ESTELLE, and UML [11, 12].

For example, Utting [12] proposed the seven-dimension taxonomy of various conceptual approaches to model-based testing, including model subject, model redundancy, model characteristics, model paradigm, test selection criteria, test generation technology, and test execution method. Cyrille et al. proposed a model-based API tester for event-driven systems named Modbat [5], which is tailored to event-driven or input/output-driven systems, which support non-blocking I/O operations, or operations throwing exceptions when communication is disrupted. It uses extended finite-state machines to model system behavior. Modbat offers a domain-specific language that supports state machines and exceptions as first-class constructs. And the model notation also handles non-determinism in the system under test, and supports alternative continuations of test cases depending on the outcome of non-deterministic operations. In [10], it proposed a model-based solution for API testing of Apache ZooKeeper[17]. Using the tool Modbat [5], it generated test cases for concurrent client sessions, and processes results from synchronous and asynchronous callbacks. It also used an embedded model checker to compute the test oracle for non-deterministic outcomes, the oracle model evolved dynamically with each new test step. It verified the complexity of ZooKeeper by using a few hundred automated unit tests, and showed how model-based testing can generate more tests, in a way that increases the diversity of action sequences that are tested and can uncover previously unknown defects. Dalal [11] introduced four case studies of applying combinational test-generation techniques in various applications. More and more researchers investigate the effective process, methods, techniques and tools to establish the model, generate tests and evaluate results in a practical approach.

However, it is yet hard to establish MBT for large-scale projects. It is quite usual that a model is effective for verifying specific algorithms or data structures, but difficult to address system level problems. To apply MBT in real industry environment faces many difficulties. One issue is to how to extract and specify software expected behavior [13]. Models are conventionally developed in two ways [14, 12]: the black box approach based on requirements specifications and the white box approach based on code structures. Requirements are mostly specified in natural languages, even using UML use case model which is widely used for industry requirements specification. It is hard to map natural language specification to formal model completely and consistently [14]. Code structure can be automatically extracted with program analysis techniques [15]. But the model derived from code represents software implementation, rather than expected behavior. Tests generated in the white-box approach cannot validate software against its requirements.

Following the SaaS concepts, software is structured into loosely-coupled modules and interfaces can be defined with well-formulated syntax and semantics. In this way, API is by nature the requirements for all the software components that are developed by individual parties and composed into the integrated system. API-based testing was thus proposed as a feasible solution of model-based test automation [16]. Blackburn [16] observed that APIs can be viewed as testable requirements specification which can be represented, understood and processed automatically by computers. Following the API-driven principle, a test automation framework and a system T_VEC was built. However, there is a risk that many hidden constraints and conditions are not visible from simple API specifications [14]. ATCloud improved the API-driven principle, by incorporating constraints and conditions into the API model.

V. Summary And Conclusion

A large number of Cloud APIs have been published, which constantly evolve online and widely influence system constructions. The paper presents the model-based testing framework ATCloud to support Cloud API testing. ATCloud aims to promote automatic API understanding and testing to facilitate continuous quality control of Cloud-based software systems. A prototype system has been implemented and exercised on cloud platforms to illustrate the process of API Understanding, API scenario modeling, massive scalable workload simulation, on-demand test resource allocation, and continuous monitoring.

Acknowledgement

This work was supported by the National Key Research and Development Program of China(2016YFB1000504) and the National Natural Science Foundation of China(61472197).

References

[1] Jacobson D, Brail G, Woods D. APIs: A strategy guide[M]. " O'Reilly Media, Inc.", pp. 1-40, 2011.

[2] Maleshkova M, Pedrinaci C, Domingue J. Investigating web apis on the world wide web[C]//Web Services (ECOWS), 2010 IEEE 8th European Conference on. IEEE, 2010: 107-114.

[3] Programmable Web, "programmable Web report," https://www.programmableweb.com, 2005 , accessed 26 January 2017

[4] Gao J, Bai X, Tsai W T. Cloud testing-issues, challenges, needs and practice[J]. Software Engineering: An International Journal, 2011, 1(1): 9-23.

[5] Artho C V, Biere A, Hagiya M, et al. Modbat: A model-based API tester for event-driven systems[C]//Haifa Verification Conference. Springer International Publishing, 2013: 112-128.

[6] Tsai W T, Bai X, Paul R, et al. Scenario-based functional regression testing[C]//Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International. IEEE, 2001: 496-501.

[7] Bai X, Li M, Huang X, et al. Vee@ Cloud: The virtual test lab on the cloud[C]//Proceedings of the 8th International Workshop on Automation of Software Test. IEEE Press, 2013: 15-18.

[8] Armstrong J. Programming Erlang: software for a concurrent world[M]. Pragmatic Bookshelf, 2007.

[9] Kamboj R, Arya A. OpenStack: open source cloud computing IaaS platform[J]. International Journal of Advanced Research in Computer Science and Software Engineering, 2014, 4(5).

[10] Artho C, Rousset G. Model-based API Testing of Apache ZooKeeper. //ICST, 10th IEEE International Conference on Software Testing, Verification and Validation, IEEE, 2017.33.

[11] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. 1999. Model-based testing in practice. In ICSE '99 Proceedings of the 21st international conference on Software engineering (Los Angeles, CA, USA, May 16 - 22, 1999). ACM New York, NY, USA, 285-294.

[12] Utting, M., Legeard, B., Pretschner, A. 2006. A taxonomy of model-based testing. Department of Computer Science, University of Waikato.

[13] Dwyer, M. B., Elbaum, S., Goddard S. A Pervasive Software Validation Approach for Next Generation Avionics Systems. Available at: http://chess.eecs.berkeley.edu/hcssas/papers/Dwyer-position.pdf.

[14] Denger, C., Mora, M. M. 2003. Test case derived from requirement specifications. Technical report. Fraunhofer IESE, Germany.

[15] Bai, X., Liu, T. 2008. SyncTest: a Tool to Synchronize Source Code, Model and Testing. In Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE'2008) (San Francisco, CA, USA, July 1 - 3, 2008). 723-728.

[16] Blackburn, M., Busser, R., Nauman, A. 2004. Why model-based test automation is different and what you should know to get started. In International Conference on Practical Software Quality and Testing. 212-232.

[17] Junqueira F, Reed B. ZooKeeper: distributed process coordination[M]. " O'Reilly Media, Inc.", 2013.