Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 13, no. 8, December 2017, pp. 1257-1267
DOI: 10.23940/ijpe.17.08.p8.12571267

SDN-based Approach to Generating and Optimizing Test Path for Cloud Application

Liqiong Chen[a,b], Yunxiang Liu[a], Guisheng Fan[c,*]

[a]*Department of Computer Science and Information Engineering, Shanghai Institute of Technology, Shanghai 200235, China*
[b]*Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 201112, China*
[c]*Department of Computer Science and Engineering East China University of Science and Technology, Shanghai 200237, China*

Abstract

With the intensive and large-scale development of cloud computing, software testing has become one of the most important problems. How to evaluate and assess the testing process of a cloud computing system is a key to the deployment and use of it. This paper proposes a method to generate and optimize test paths of cloud application based on SDN, which separates the cloud service testing process from the underlying execution logic, thus improving the scalability of the testing process. The finite state machine (FSM) is used to establish the formal description language of the cloud application testing process, and is also used to model the basic elements, such as cloud services, jobs, test cases and cloud applications, and construct a test model for cloud application. The related theory of FSM is used to analyze the effectiveness and correctness of cloud application test model. Based on the actual mapping of the model, the generation method of test path is also given. In addition, we analyze the coverage of test cases and propose the test path optimization methods to improve the test efficiency. The specific examples and simulations show that this method can simplify the design and analysis of the cloud application testing process and effectively improve the generation of test paths.

Keywords: cloud computing; software defined network; testing; finite state machine; verification

(Submitted on September 30, 2017; Revised on November 1, 2017; Accepted on November 17, 2017)

1. Introduction

Cloud computing is a new type of computing model that provides computing and storage services to users; it has become the developing trend in the IT industry [18]. Cloud computing effectively integrates network resources and uses virtualization technologies to separate physical and virtual services and improve resource utilization [5]. Elasticity is the key feature of cloud computing, which can support resources to elastically expand, quickly configure, flexibly add and remove, and make full use of cloud resources to reduce the cost of cloud services providers and users [19]. Currently, there are a lot of cloud computing products, but as an emerging technology and field, there is still a lack of the related testing and evaluation methods. How to evaluate and assess the cloud computing testing process is a key to the deployment and use of it.

However, how to generate and optimize the test path for cloud computing system is a complicated issue. This is because varieties of application services deployed under cloud computing system have complicated provisioning, configuration, and requirements. These features require the modeling language to own the powerful expressivity and scalability of characterizing the testing process of cloud application and providing the means to compose the basic elements models. Moreover, testing process of the cloud computing system may involve multiple resources, and the flexibility, concurrency and uncertainty of resource scheduling process make model verification be complicated. The existing testing method is difficult to directly use in large-scale cloud application system. In addition, the addition and modification of testing function will become extremely difficult, which cannot effectively test all types of situations that occur with cloud applications. The separation of network control and the data forwarding is realized by the centralized controller software platform, and then achieve the programmable control of the underlying hardware and flexible deployment of network resources on-demand.

* Corresponding author.
E-mail address: gsfan@ecust.edu.cn

SDN is derived from Clean Slate by University of Stanford. It is a novel network architecture that can separate out network control functions of the underlying devices and centrally deploy them into the controller. The SDN switches of the underlying devices are just for data forwarding. With SDN, applications can regard the network as a logical entity. Thus, enterprises can obtain unprecedented programmability, automation, and network control [23]. In addition, SDN also provides a set of application programming interfaces(APIs) to simplify the implementation of common network services such as routing, access control, bandwidth management et al. As a consequence, SDN creates plenty of chances to help enterprises build more deterministic, innovative, manageable and high scalable data centers. The leading technology of SDN is based on the OpenFlow protocol [22,24], which has already been a standard for SDN and been used in varieties of network and networking products. SDN provides a high degree of scalability for software testing. With the expansion of testing size and testing rules of cloud computing, SDN can encapsulate software testing process and parameters, thus improving the efficiency of software testing. Using SDN to separate test functions and test environment deployments from the execution code of cloud application will make the system easier to maintain and reduce the complexity of cloud application testing, allowing the programmers to focus on the business logic.

SDN based testing framework of cloud application can enhance the test flexibility and code reusability. However, the requirements analysis of cloud applications often presents fuzziness and ambiguity, and it is difficult to conduct rigorous semantic analysis and reasoning on complex cloud systems. While many development tools can help designers discover the syntax errors of cloud services, their underlying logic errors cannot be completely eliminated. In order to automatically generate test cases, we need to formally model the system behavior. Finite state machine (FSM) has a mature theoretical foundation, and can be designed, manipulated and analyzed by using the theory of formal language and automaton [16]. It is the most commonly used software testing model. The approach to generating test cases based on FSMs assumes that the given model is identical with user requirements and the goal of the testing is to check if the implementation conforms to the given model. It gives the algebraic representation of FSM and its corresponding testing theory, and employs regular expressions to describe test paths (abstract test cases) of cloud computing, which simplifies the test analysis and helps to conduct formal deduction.

This paper investigates how to generate and optimize the test case for cloud computing. FSM is adopted as an underlying formalism, which provides a formalism to depict the internal logic and behavior of the SDN enabled testing process as well as the sufficient analysis ability for supporting the verification of testing process. Compared with the existing works, our contributions are shown as follows: (1) SDN is introduced into testing process of cloud application, with the powerful bandwidth control capability of SDN, which can separate the cloud service testing from the underlying service, thus improving the scalability of the testing process. (2) Finite state machine is used to accurately describe different components and their relationships in the SDN enabled testing process of cloud application. The interface matching is used for merging basic models, thus the testing model can be constructed. (3) The related theory of FSM is used to analyze the effectiveness and correctness of cloud application test model. Based on the actual mapping of the model, the algorithm for generating test path is given. In addition, we analyze the coverage of test cases and propose test path optimization methods to improve the test efficiency.

This paper is organized as follows. Section 2 introduces the related works, and the following section introduces SDN-based cloud application testing framework, Section 4 introduces the test model of cloud application, the analysis of the model and the corresponding algorithm are shown in the Section 5, Section 6 is the simulation experiment, the conclusion and future work is presented in Section 7.

2. Related Work

In this section, some related works which are directly or indirectly of interest to our work are introduced.

In previous research work, a lot of research has been done to investigate the impact of the used test cases on the effectiveness of fault localization, and among them, various test suite reduction approaches are proposed in [7,8,13,14,15]. Harrold et al. [13] proposed a heuristic algorithm (referred to as HGS) to reduce the size of a test suite. They categorized test cases in the test suite according to the importance of coverage requirements of corresponding test cases, and select a representative set of test cases that satisfies all of the requirements as the original test suite, thus potentially minimizing the test suite. Chen et al. [7] proposed another Heuristic algorithm called Heuristic GRE. Hao et al. [14,15] assumed that redundant test cases might result in a negative effectiveness of fault localization and proposed to use traditional test-reduction strategies before applying fault localization techniques. They proposed a test-reduction approach (including three strategies) to reduce the similarity and redundancy test cases based on the execution traces of the test set to improve the effectiveness of fault localization. Patrick Daniel et al. [8] developed a novel SFL tool with test case preprocessor to eliminate test cases with contradicting, duplicated or other noisy spectra.

Software defined network is a new paradigm which separates the control and data (or forwarding) planes to the independent devices. It is the key technology that can improve the scalability, manageability, controllability and dynamism of cloud [25]. Because of these advantages, SDN-based cloud has been introduced in many research. Yen et al. [26] propose an SDN-based cloud computing environment by open source OpenFLow switch and controller packages. And the functionality of OpenFLow controller is extended to provide mechanisms of load balancing, power-saving, and monitoring. The authors of [21] focus on ad hoc networks, presenting several designs for SDN-based mobile cloud architectures. Several instances of the proposed architectures based on frequency selection of wireless transmission are also introduced. Gharakheili et al. [11] developed an architecture that is composed of a cloud-based front-end user interface and SDN-based APIs in the back-end. A QoS-guaranteed approach is presented in [2] for the purpose of bandwidth allocation by using Open vSwitch based on SDN. And the work presented by [6] survey the security on SDN. Although the security of SDN is very important, it is not the focus of this paper. SDN can provide powerful network control capability that the real-time network state like network traffic and bandwidth can be obtained. Therefore, in this paper, SDN is introduced to improve the performance of software testing for cloud application.

Formal method techniques have been adopted in many research for its mathematically foundation of providing theoretically sound and correct formalism. Reference [20] proposed a powerful genetic algorithm for optimizing the task scheduling solutions. A behavioral modeling approach with model checking techniques is presented by them. Other works about formal semantic analysis of cloud computing are proposed in [1,4,9,12]. [12] develops a scalable stochastic analytic model for performance quantitation of Infrastructure-as-a-Service (IaaS) Cloud. Reference [1] presents a method to measure and analyze different strategies for optimizing energetic cost in cloud environments. Abstract State Machine is used to obtain a precise model for the client-server application of cloud systems [4]. [9] proposed an adaptive resource scheduling strategy for cloud computing based on reflection mechanism and Petri nets. In contrast, the task scheduling model of SDN enabled Hadoop is proposed in this paper. And the underlying formalism is Petri net, which provides means to observe behaviors of basic components and to describe their interrelationships [1,10].

3. SDN-based Cloud Application Testing Framework

3.1. Framework

As the cloud application in the cloud computing finds a suitable service for each job, the service will be executed on a physical node. Cloud services are the basic components of cloud applications, and each cloud application can realize the function of one or more jobs. Due to the distribution and heterogeneity of cloud computing, the occurrence of service failure is inevitable. The service failure means that the operation behavior of service node's software/hardware cannot meet the requirements of user, which is a dynamic feature of cloud computing. In order to efficiently test cloud applications, this paper presents a SDN based testing framework for cloud application software, which is shown in Figure 1.

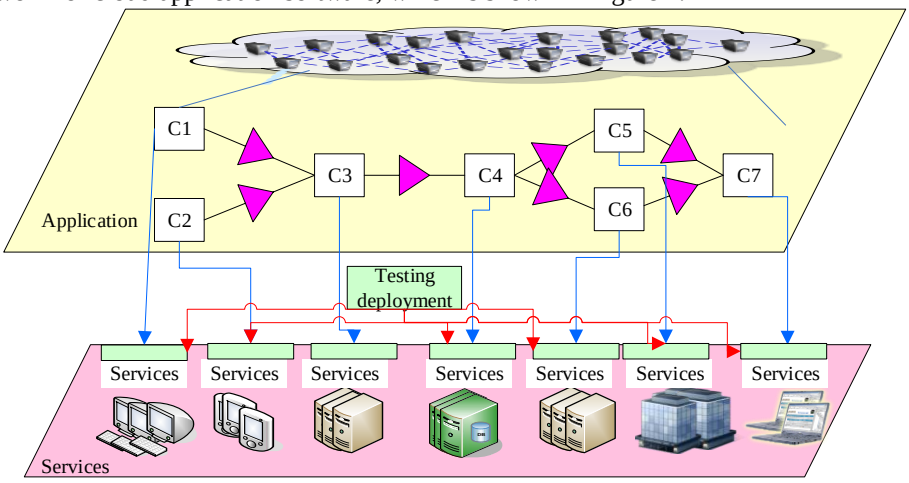


Figure 1. Testing framework

The framework includes the infrastructure layer, control layer and application layer by using the idea of SDN. We add the test deployment layer between the service layer and the application layer for realizing the test deployment and implementation of cloud applications.

3.2. Testing requirement of cloud application

We will give the testing requirement of cloud application from the view of the test path generation and optimization.

Definition 1: The testing requirement of cloud applications is a 5-tuple: $\Xi = \{Job, WS, Tcase, JT, JW\}$: where

1) *Job* is the set of limited jobs, and each job can realize a function of cloud application.

2) *WS* is a set of limited services (resource services are software, cloud applications can access the physical nodes and databases through resource services), the services are atomic services.

3) *Tcase* is the initial set of test cases, each test case $tcase_i$ has a set of testing service TW_i, which is the testable service of $tcase_i$.

4) *JT* is a relational function between jobs. There are three types of relations between jobs: sequence, choice and parallel. If $JT(Job_i, Job_j)=\bullet$, it means that the relationship between Job_i and Job_j is sequence, Job_i is called the forward job of Job_j, and Job_j is the afterward job of Job_i.

5) *JW* is the mapping relationship between job and service, $\forall Job_i \in Job$， $JW_i=\{WS_k, WS_f, \ldots\}$ is the set of available service of Job_i.

Definition 2. The five tuple $\Psi= (S, \Sigma, \delta, S_0, F)$ is a finite state machine:

1) S is a state set, Σ is a set of symbols on the transition;

2) δ is a state transition function from $S\times\Sigma\rightarrow S$, $\delta(S_l, t)=S_2$ indicates that when symbol in the transition is t under the state S_l, the system can reach the state S_2;

3) S_0 is the only initial state;

4) F is the set of termination states.

In software systems, the state of the FSM is usually represented by state variables with the discrete data types. When it gets an input parameter, it will transfer from the current state to another state or remain in its current state. For the FSM model Ψ, the rule it selects for the next state is shown on the migration function </s>. If Ψ gets the input parameter t in state S, the system will reach state $\delta(S, t)$. We can use the finite state machines to describe the testing process of cloud applications. For example, S represents the state of the model (cloud service, component), t represents the possible operations that can be executed by t.

In order to better describe the hierarchical and complex behavior of cloud applications, we have extended the basic model of FSM.

Definition 3. A four tuple $\Psi = (S, \Sigma, G, \delta)$ is a finite state machine of cloud application (*CFSM*):

1) $S=\{S_0, S_f, \ldots\}$ is the set of state, S_0 and S_f are the unique initial and termination state. State S can also be the initial state and the termination state of another CFSM. It can assume that if Ψ_2 is the initial state / termination state of Ψ_1, then Ψ_1 must have the initial / termination state of Ψ_2.

2) Σ is a set of symbols on the transition. *null* is defined a transition, if there is not any firing transition under state S_1, the system will fire *null* transition to reach the termination state. There is also a special transition in the model, which represents the transition from the initial state to the termination state of the submodel (we call it as the virtual transition).

3) δ is a state transition function of $S\times\Sigma\rightarrow S$.

4) G is the firing condition of transition. We can assume that the firing condition of virtual transition is empty. The normal transition can be fired when the firing condition is true.

5) Tc is an attribute of a state, which is a subset of the test case set.

CFSM is an extension of *FSM*. Because *CFSM* includes the initial and termination states of *CFSM*, *CFSM* can describe the hierarchy structure of cloud applications. In addition, we can achieve the control of test path selection by controlling the firing conditions of transition.

4. Modeling Testing Process of SDN-based Cloud Application

We will use *CFSM* to model testing process of cloud application.

(1) Modelling the Testing Process of Cloud Service

The model of the testing process of cloud service WS_i by using *CFSM* is shown in Figure 2 (a), where state $S_{0,i}$ indicates that the service WS_i is in the state of waiting for test. If the current set of test case has a test case that can be used to test the service WS_i, then fire the transition $t_{0,i}$. Else fire the *null* transition t_{null}. States $S_{t,i}$, $S_{f,i}$ represent the testing process and the termination state.

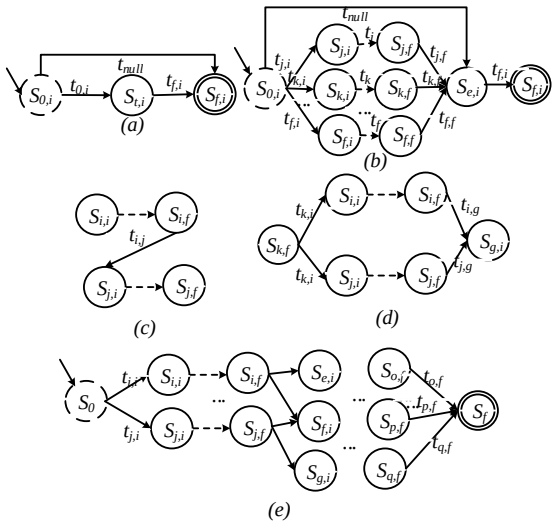


Figure 2. Testing process of cloud service

(2) Modelling Testing Process of Job

The testing process of Job_i is: it gets a set of test cases, then uses the set of test cases to test the available services of the job. If no available services of Job_i can be tested by the current set of test cases, then do the operation of null transition. The specific test model is shown in Figure 2(b). State $S_{0,i}$ indicates that job Job_i is in the state of waiting for testing. If $\in JW_i$, then introduce the test model of service (the internal process of sub-model is denoted by virtual transition tj), and there is $\delta(S_{0,i}, t_{j,i}) = S_{j,i}$. $S_{j,i}$ is the initial state of the service WS_j, Transition $t_{j,}$ is used to start the testing process of service WS_j. State $S_{e,i}$ indicates that the testing process of a service in job Job_i has finished. Similarly, if $\delta(S_{0,i}, t_{j,i}) = S_{j,i}$. According to the hierarchy of the cloud application, the test model of job is in the middle layer. Therefore, the test model of job can include some test models of service. The test model of job is also a submodel of cloud application model.

(3) Modeling the Basic Relationships Between Jobs

The modeling process of the basic relationships between jobs is shown in Figure 2 (c) - (d). The sequence relationship is shown in Figure 2 (c), the Job_j can be tested only when all the forward jobs of the job Job_i have been tested. Therefore, for the sequence relationship $Job_i \bullet Job_j$, it needs to introduce the transition $t_{i,j}$ to transfer the test case.

The choice relationship and the parallel relationship are shown in Figure 2 (d). We can set the corresponding forward and backward jobs as Job_k, Job_g. We can control the firing function G of the transition to achieve the choice or parallel relationship. For the choice relationship $Job_i + Job_j$, the test set can only test for available services of one job Job_i, the corresponding transition is fired to transfer the set of test case to the initial state $S_{i,i}$ of Job_i. If the test set can be used to test both jobs, then fire the selection transition to select a job. If the test of a job is completed, we can start the test of job Job_g. For the parallel relationship $Job_i | Job_j$, we can fire the transition to do the testing of Job_i, Job_j, and the job Job_g can be started when both tests are completed.

(4) The Testing Process of Cloud application

The testing process of cloud application is: if the system gets a set of test cases, we can order the sequence of test cases according to their relationships between the jobs. The previous job can be firstly tested. The system will be in the termination state of testing process until all the jobs are tested according to the business process. The specific test model is shown in Figure 2 (e), where the states S_0 and S_f represent the initial state and the termination state of test in the cloud application. Each job is a sub model in the cloud application layer. We can continue to extend the model based on the actual requirement.

5. Model Analysis and Enforcement Algorithm

The transitions fired from state S_i to state S_j in the test model of cloud application Ψ are constituted in a sequence $\zeta = t_1 \Theta\, t_2 \ldots \Theta\, t_n$, ζ and is called a transition sequence from S_i to S_j. Path(S_i, S_j)={ζ|ζ ζ is a transition sequence from S_i to S_j} and is the set of firing transitions from S_i to S_j.

Definition 4: Let ζ be the firing sequence from state S_i to S_j in the model *FDM*, if:

(1) $S_i=S_0$, then ζ is the transition path of Ψ;

(2) If $S_i=S_0 \wedge S_j \in F$, then then ζ is the full path of Ψ.

The transition path is a firing transition sequence from the initial state. The full path is the firing transition sequence from the initial state to the termination state. A test path can be a transition path or a full path. We will analyze the effectiveness and correctness of constructed model based on the transition sequence and path of test model.

Theorem 1: The testing requirement of cloud application is Ξ, the corresponding test model is Ψ, then $\forall WS_i \in WS$, if $\exists Tcase_j \in Tcase$, which makes $WS_i \in TW_j$, then Ψ has a transition path ζ, which makes $t_{i,i} \in \zeta$.

Proof: We can assume that WS_i is an available service of job Job_k.

Because $Tcase_j \in$ Tcase, which makes $WS_i \in TW_j$, that is, the service WS_i can be tested by the test case $Tcase_j$.

Because the set of test case in the requirement Ξ is *Tcase*. So the initial state S_0 of test model of cloud application has the attribute $Tc(S_0)= Tcase$.

We will prove the theorem based on two cases: Job_k has the forward job and Job_k does not have the forward job.

(1) If Job_k does not have the forward job, we can get from the modeling process of test model of cloud application, there is a firing path ζ_1, which makes $t_{k,i} \in \zeta_1$. That is, the system will fire the test of Job_k, ie, $S_0\zeta_1 S_{k,i}$. Because WS_i is the available service of Job_k, Job_k will fire the transition $t_{i,i}$ to deploy the test case to the service WS_i. According to the principle of the test model of job, the system can fire the test of the service WS_i, that is, the initial state $S_{i,\ I}$ of service WS_i has the attribute $Tc(S_{i,i})=Tc(S_0)= Tcase$. Because $\exists Tcase_j \in Tcase$ makes $WS_i \in TW_j$. So the model will fire the migration $t_{i,i}$ to make the service be in the test process, that is, there is firing sequence ζ_2, which makes $t_{i,i} \in \zeta_2 \wedge S_{k,i}\zeta_2 S_{t,i}$. And $\zeta=\zeta_1 \cup \zeta$, then ζ is a transition path, and $t_{i,i} \in \zeta$.

(2) If Job_k has a forward job. We can get that there is a transition path ζ_1, which makes $S_0\zeta_1 S_{k,I}$ according to induction. Job is in the initial state of waiting for test under state $S_{k,i}$. According to the proof process of (1). If Job_k has a forward job, the result is also established.

In summary, $\forall WS_i \in WS$, if $\exists Tcase_j \in Tcase$, which makes $WS_i \in TW_j$, then Ψ has a transition path ζ, which makes $t_{i,i} \in \zeta$.

Theorem 1 states that all testable services in the test model can be tested. It shows that the constructed test model of cloud application can express the relationship between test cases and services.

Definition 5: (Algebraic System) In CTFSM, the set of transition sequence π and the set of operator{*,+,•,|} form a FSM algebraic system and denoted as <π， *,+,.,|,λ， ε>, Where λ is the sequence connection identity, and ε is a zero element.

A regular expression constructed from CTFSM is a sequence of states and transitions joined by operator +、 +…l and parentheses (,). That is, the elements of a regular expression are the states and transitions in CTFSM. Operator "*" indicates the exponential operation. t* indicates that *t* can not appear in the regular expression, appear one or more times. Operator "." is the sequential connection operation, operator "+" is the choice operation (Or). The operator "|" represents the parallel operation. For example, $t_1.(t_2+t_3)$ represents $t1 \cdot t2$ or $t1 \cdot t3$. The priority of operator "+" is higher than operator "+" and "|", and the priority of sequential connection operation "." is higher than "+". In addition, these operators are left-associative. A CTFSM algebraic system can be used to represent a test path for a cloud application test model. The cloud application test model migration sequence is a test path. For complex systems, the larger the choice case set, the greater the overhead of switching and managing between use cases. Therefore, it is necessary to minimize the size of the selected use case set while ensuring the correct operation of the system. The following analysis of the inherent correlation between test cases.

The algebraic system of CTFSM can be used to represent a test path for test model of cloud application. The transition sequence of test model of cloud application is a test path. Firstly, we initialize the set of test case. For complex systems, the larger the set of test case, the greater the overhead of switching and managing between use cases. Therefore, it is necessary to minimize the size of the selected set of test case set while ensuring the correct operation of the system. We will analyse the inherent correlation between test cases.

Definition 6: Let Ξ be the testing requirements model of cloud application, $\forall tcase_i$, $tcase_j \in Tcase$, ($i \neq j$) are two test cases of the system, if:

(1) If $TW(tcase_i) \cap BT(tcase_j) \cap WS = \varnothing$, then $tcase_i$, $tcase_j$ are irrelevant in the set of service WS, denoted by $tcase_i \infty_0 tcase_j$, otherwise, it is denoted by $tcase_i \infty tcase_j$.

(2) If $TW(tcase_i) \cap WS \subset TW(tcase_j) \cap \mathrm{WS}$, then $tcase_j$ can cover $tcase_i$ on the set of service WS, denoted by $tcase_j \nabla tcase_i$, otherwise, it is denoted by $tcase_j \Delta tcase_i$.

(3) If $TW(tcase_i) \cap WS = TW(tcase_j) \cap WS$, then $tcase_j$ is equivalent to $tcase_i$ on the set of service WS, it is denoted by $tcase_j \approx tcase_i$.

The two test cases are irrelevant, it means that their testable set of services do not intersect. Let *Tcase* be a set of test case for testing cloud application, and we can analyze the relationship between test cases.

Definition 7: Let Ξ be the testing requirements model of cloud application, $tcase_i$ is a test case, $WS_f \subseteq WS$ is the set of service: $\mathrm{Cov}(tcase_i, \mathrm{WSf}) = \frac{|\mathrm{TW}(tcasei) \cap \mathrm{WSf}|}{|\mathrm{WSf}|}$ is called the coverage ratio on the set of service WS_f.

The larger the coverage ratio $\mathrm{Cov}(tcase_i, WS_f)$, the higher proportion of testable services that test case $tcase_i$ on the set of service WS_f. The test process has a situation where several test cases can be used to test the current service. In addition, there is some overhead in the switch and manage between test cases. Therefore, it is necessary to guide the generation of test paths based on the relationships and coverage between test cases.

Definition 8: Let Ξ be the testing requirements model of cloud application and Ψ be the corresponding test model. The test path generation and optimization (CPO) algorithm is as follows:

Step 1: Filtering the set of test case

Initialization: The alternative set of test case *OTC* is initialized to *Tcase*, the set of service to be tested $CWS=WS$:

(1) If $CWS==\varnothing$, then end this step and output *OTC*, and go to Step 2. Otherwise, $\forall\ tcase_i \in OTC$, we can set the current test $tc = tcase_i$ and do Step(2).

(2) If $\exists tcase_j \in OTC$, which makes $tcase_j \nabla tcase_i \vee tcase_j \approx tcase_i$ on the set of service *CWS*, then $OTC = OTC\text{-}tcase_i$.

(3) If $\exists tcase_j \in OTC$, which makes $tcase_i \nabla tcase_j$ on the set of service *CWS*, then $OTC = OTC\text{-}tcase_j$.

(4) Updating $CWS=CWS\text{-}TW(tcase_j)$ and go to Step (1).

Step 2: Model-based test path generation and optimization

Initialization: The set of test case $TC = OTC$, the current set of service *DW* to be tested is empty, the dynamic scheme *DP* is empty, and the current state *S* is initialized to S_0.

(1) If $S==S_f$, then end the step and output DP, otherwise do Step (2).

(2) Randomly selecting the enabled transition *t* under state *S*, if $t==t_{k,i}$, $k \in R$, then $DW=JW(Job_k)$ and do Step (3).

(3) If $DW==\varnothing$, then turn to Step (4), otherwise $\forall\ tcase_i \in TC$, we compute $Cov(tcase_i, DW)$. If the largest value of $Cov(tcase_i, DW)$ in *TC* is the test case $tcase_k$, then $DP=DP \cup (tcase_k, WS_j)$, where $WS_j \in TW(tcase_k) \cap DW$ and update $DW= DW\text{-}\ \mathrm{TW}(tcase_k) \cap DW$, then do Step(3).

(4) Updating the current state $S=\delta(S, t)$ and do Step (1).

Step 3 Updating the set of testable service by using the test case

Initialization work: Initialing tested service set of test case tcasei : $DTW_i = \varnothing$, the current test scheme $CDP = DP$.

(1) If CDP == ∅, then the end the step and output DTW, otherwise do Step (2);

(2) arbitrarily take an element in *CDP*: (*tcasei*, *WSk*), update $DTW_i = DTW_i \cup \{WS_k\}$;

(3) Update *CDP*: $CDP = CDP - \{(tcase_i, WS_k)\}$, and do Step (1);

Step 4 Generating the test path

Initialization: The current state *S* is initialized to S_0, test path $P = \varnothing$, test job path $JP = \varnothing$, test service path $WP = \varnothing$.

(1) If $S==S_f$, the process is terminated and output *P*, *JP*, *WP*. Otherwise, it will do Step (2);

(2) Firing the transition *t* in place under the state *S*, $P = P \cup t$;

(3) If *t* is the fired transition of job Job_i, $JP = JP \cup Job_i$;

(4) If *t* is the fired transition of service WS_j, then $WP = WP \cup WS_j$;

(5) Updating the current state $S = \delta(S, t)$, and do Step (1).

We can filter the set of test case by using the above strategy. And the set of DTWi≠∅ test cases from Step 3 is used as input for the test model. At the same time, *TW* of these test cases are updated using Step 3. Through the algorithm *CPO* we can optimize the test suite. Because the test path is a transition sequence of the test model. So the set of input test case and its set of tested service will directly affect the test path. Our algorithm *CPO* can optimize the test path for cloud applications.

Table 1. Test path generation and optimization(CPO)

Input: Requirement model Ξ and test model Ψ
Output: Test path *OP*
Filter_Tcase(case, WS)//Step 1: adjusting the alternative set of test case
Step1: Initializing *OTC=Tcase*, *CWS=WS*
Step2:if (*CWS*==∅), then output(*OTC*)
Step3:Else foreach $tcase_i$ in *OTC*
Step4:{ $tc= tcase_i$;
Step5:if $\exists tcase_j \in OTC$, which makes $tcase_j \nabla tcase_i \vee tcase_j \approx tcase_i$ on the set of service *CWS*, then $OTC= OTC\text{-}tcase_i$;
Step6:*else if* $\exists tcase_j \in OTC$, which makes $tcase_i \nabla tcase_j$ on the set of service *CWS*, then $OTC= OTC\text{-}tcase_j$;
Step7:$CWS=CWS\text{-}TW(tcase_j)$;
Step8:}}

Co_scheme ()//Step 2: Model-based test scheme generation and optimization
Step9:{*TC*= Filter_Tcase(*Tcase*, *WS*);
Step10:*DW=DP=*∅;
Step11:$S=S_0$;
Step12:While($S \neq S_f$)
Step13:{Randomly selecting the enabled transition *t* under state *S*, if($t==t_{k,i}$), $k \in R$, then $DW=JW(Job_k)$;
Step14:While (*DW*≠∅)
Step15:{foreach $tcase_i$ in *TC* computing $Cov(tcase_i, DW)$;
Step16:If the largest value of $Cov(tcase_i, DW)$ in *TC* is the test case $tcase_k$, then
Step17:{foreach WS_j in $TW(tcase_k) \cap DW$
Step18:$DP=DP \cup (tcase_k, WS_j)$;
Step19:$DW= DW- TW(tcase_k) \cap DW$;}
Step20:}
Step21:$S=\delta(S, t)$;}
Step22:Output(*Dp*);
Step23: }

Update_tw ()//Step3: Update the test case's testable service set
Step24:{ *CDP= Co_scheme* (); $DTW_i=\emptyset, i=1,...,|Tcase|$;
Step25:While(*CDP* ≠∅)
Step26:{Randomly taking an element in *CDP*: $(tcase_i, WS_k)$, update $DTW_i = DTW_i \cup \{WS_k\}$;;
Step27: $CDP=CDP\text{-}\{(tcase_i, WS_k)\}$
Step28:Output(*CDP*);
Step29:$TW_i=DTW_i$, $i=1,...,|Tcase|$;
Step30: }

Co_Path ()//Step 4: Generate test path
Step31:{ *Update_tw* ();
Step32:*P=JP= WP=*∅;
Step33:$S=S_0$;
Step34:While($S \neq S_f$)
Step35:{ Firing transition *t* in arbitrary state *S*, $P = P \cup t$;
Step36: If *t* is the fired transition of job Job_i, then $JP = JP \cup Job_i$;
Step37: If *t* is the fired transition of service WS_j, then $WP = WP \cup WS_j$;
Step38: $S = \delta(S, t)$;}
Step39: Output *P*, *JP*, *WP*;
Step40:}

6. **Experiments**

Example 1: The execution process of scientific computing cloud Ligo application app1 is: $Job_1 \bullet (Job_2 \bullet Job_3) | Job_4) \bullet (Job_5 + Job_6)) \bullet Job_7$. Let the cloud environment contain 21 cloud services which are related with app1, 12 test cases. Randomly generate the attributes and relationships, which is shown in Table 2.

According to Table 2, we can get that the coverage ratio of each test case under service *WS* is {14.3%,9.5%,9.5%,19%,19%,19%, 14.3%,9.5%,19%,9.5%,19%,9.5%}. According to CPO algorithm, we can calculate the corresponding test scheme is {($Tcase_{5,}$ WS_1), ($Tcase_{5,}$ WS_2), ($Tcase_{11,}$ WS_3), ($Tcase_{4,}$ WS_4), ($Tcase_{4,}$ WS_{21}), ($Tcase_{6,}$ WS_5), ($Tcase_{11,}$ WS_{10}), ($Tcase_{11,}$ WS_{20}), ($Tcase_{7,}$ WS_6), ($Tcase_{7,}$ WS_7), ($Tcase_{7,}$ WS_8), ($Tcase_{5,}$ WS_{11}), ($Tcase_{12,}$ WS_{12})}. The test model of cloud application *app1* is shown in Figure 3. The model mainly describes the execution process of app1 and the corresponding testing process of the job and cloud service. The related tools of FSM can be used to verify the properties of the model. We can also get that the execution process is correct based on the state space of constructed model, that is, there is a transition sequence which can make all the jobs realize the corresponding test. Mapped into the constructed model Ω, there is a full path which can make the model reach the state S_f.

Table 2. The relationship between job, test case and service

Job	**JW**	**Tcase**	**TW**	**Tcase**	**TW**
Job_1	WS_1, WS_2	$Tcase_1$	WS_{13}, WS_{14}, WS_{15}	$Tcase_8$	WS_{16}, WS_{17}
Job_2	WS_2, WS_3, WS_4	$Tcase_2$	WS_{16}, WS_{17}	$Tcase_9$	WS_{18}, WS_{19}, WS_{20}, WS_{21}
Job_3	WS_5, WS_{21}	$Tcase_3$	WS_{18}, WS_{19}	$Tcase_{10}$	WS_{16}, WS_{17}
Job_4	WS_{10}, WS_{20}, WS_2, WS_3	$Tcase_4$	WS_4, WS_{19}, WS_{20}, WS_{21}	$Tcase_{11}$	WS_{10}, WS_{20}, WS_2, WS_3
Job_5	WS_6, WS_7, WS_8	$Tcase_5$	WS_1, WS_2, WS_{11}, WS_{21}	$Tcase_{12}$	WS_{12},WS_{20}
Job_6	WS_9, WS_{10}	$Tcase_6$	WS_{10}, WS_{11}, WS_2,WS_5		
Job_7	WS_{11}, $WS1_2$	$Tcase_7$	WS_6, WS_7, WS_8		

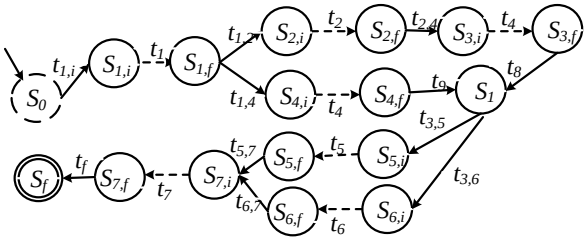
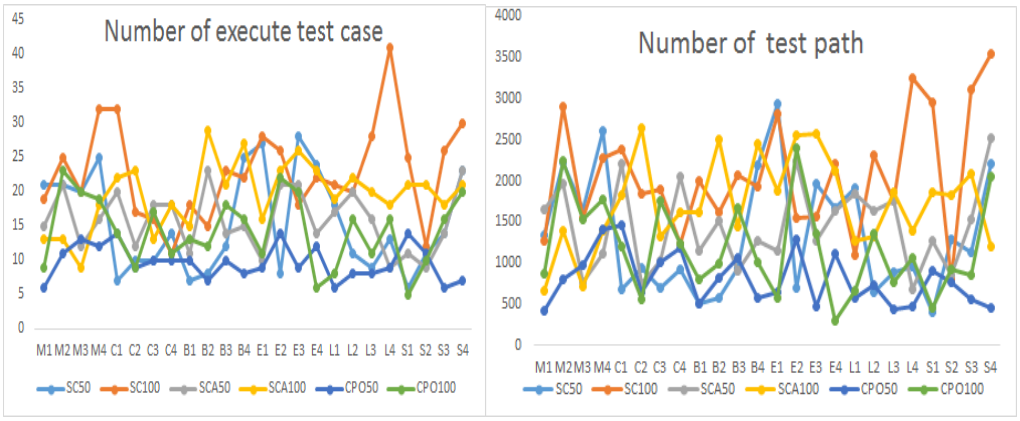


Figure 3. Test model of app1



(a) (b)

Figure 4. Simulation results of Experiment 1

In order to effectively evaluate the effectiveness of the method, this paper designs an experiment to evaluate the effectiveness of the method. The experiment is conducted based on the six real business processes described in [17]: Montage, Cybershake, Broadband, Epigenomics, LIGO Inspiral Analysis, and SIPHT, which are used to analyze the test model of cloud application and the test path optimization process. Due to the lack of a unified standard service library, we use Cloudsim to construct the library of service, job, test case and their relationships. One of the benefits of using Cloudsim to construct cloud library is that we can set the different parameters according to the actual requirement.

The purpose of Experiment 1 is to analyze the effectiveness of the proposed method. The specific experimental steps are as follows:

(1) There are 24 cloud applications in (Montage, Cybershake, Broadband, Epigenomics, LIGO Inspiral Analysis and SIPHT). Each business process has 4 cloud applications and the job of each cloud application has 3-10 services, thus forming a set of available services.

(2) Taking 50 ,100 sets of test cases and randomly generating the set of testable service by using the test cases.

(3) Adopting a random selection algorithm (SC), static coverage-based algorithm (Test cases with high coverage has the higher priority), the CPO method is used to construct the test suite.

(4) Computing the size of the used test cases.

(5) Computing the number of possible test paths under each test suite.

The simulation results of Experiment 1 are shown in Figure 4. According to the results, we can draw that: (1) CPO algorithm can optimize the test suite and generate test path for cloud applications with different business processes. (2) The optimization results of algorithm are also related to the initial set of test case and its attributes. (3) Compared with the other two algorithms, the algorithm proposed in this paper has the obvious optimization results.

7. **Conclusions**

In this paper, SDN is introduced into the testing process of cloud application. With the powerful bandwidth control capability of SDN, it provides a formalism to depict the internal logic and behavior of the SDN enabled testing process as well as the sufficient analysis ability for supporting the verification of constructed system. Based on this, a test model is proposed, which is used to accurately describe different components and relationships between these components of SDN enabled testing process. The operational semantics and related theories of FSM are to verify the dynamic behaviors of the established model. Via simulation experiments, results show that the proposed method can improve the efficiency of the testing process.

Acknowledgements

This work is partially supported by the NSF of China under grants No. 61702334 and No. 61772200, Shanghai Pujiang Talent Program under grants No. 17PJ1401900. Shanghai Municipal Natural Science Foundation under Grants No. 17ZR1406900 and 17ZR1429700. Educational Research Fund of ECUST under Grant No. ZH1726108. The Collaborative Innovation Foundation of Shanghai Institute of Technology under Grants No. XTCX2016-20.

References

1. M. Alansari and B. Bordbar, “Modelling and analysis of migration policies for autonomic management of energy consumption in cloud via Petri nets,” *in 2014 International Conference on Cloud and Autonomic Computing g*, pp. 121-130, September, 2014.
2. A. Akella and K. Xiong, “Quality of Service (QoS)-Guaranteed Network Resource Allocation via Software Defined Networking (SDN),” *in Proceeding of the 12th International Conference on Dependable, Autonomic and Secure Computing*, pp.7-13, August, 2014.
3. M. Alansari and B. Bordbar, “Modelling and Analysis of Migration Policies for Autonomic Management of Energy Consumption in Cloud via Petri-nets,” Mathematical Problems in Engineering, pp. 121 - 130, 2013.
4. P. Arcaini, R. Holom and E. Riccobene. “ASM-based formal design of an adaptivity component for a Cloud system,” *Formal Aspects of Computing*, vol. 28, no.4, pp. 567 - 595, 2016.
5. K. Alhazmi, A. Shami and A. Refaey, “Optimized provisioning of SDN-enabled virtual networks in geo-distributed cloud computing datacenters,” *Journal of Communications and Networks*, vol. 19, no. 4, pp. 402–415, 2017
6. S. Bian, P. Zhang and Z. Yan, “A Survey on Software-Defined Networking Security,” *in Eai International Conference on Mobile Multimedia Communications.Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*(ICST), pp. 190-198, June, 2016.
7. T. Y. Chen and M. F. Lau, “A new heuristic for test suite reduction”, *Information and Software Technology*, vol. 40, no.5, pp. 347-354, 1998.
8. P. Daniel and K. Y. Sim, “Spectrum-based fault localization tool with test case preprocessor,” *in IEEE Conference on Open Systems (ICOS)*, pp.162-167, December 2013.
9. G. Fan, H. Yu and L. Chen, “A formal aspect-oriented method for modeling and analyzing adaptive resource scheduling in cloud computing,” *IEEE Transactions on Network and Service Management*, vol. 13, no. 2, pp. 281-294, 2016
10. G. Fan, H. Yu, L. Chen and D. Liu, “Petri net based techniques for constructing reliable service composition,” *Journal of Systems and Software*, vol. 86, no. 4, pp. 1089-1106, 2013
11. H. Gharakheili, J. Bass, L. Exton and V. Sivaraman, “Personalizing the home network experience using cloud-based SDN,” *in Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pp.1-6, June, 2014.
12. R. Ghosha, F. Longob and V. Naikc, “Modeling and Performance Analysis of Large Scale IaaS Clouds,” *Future Generation Computer Systems*, vol. 29, no. 5, pp. 1216-1234, 2013.
13. M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for controlling the size of a test suite”, *ACM Transactions on Software Engineering and Methodology* (TOSEM), vol.2, no.3, pp. 270-285, 1993.
14. D. Hao, T. Xie, L. Zhang, and X. Wang, “Test input reduction for result inspection to facilitate fault localization,” *Automated Software Engineering*, vol. 17, no.1, pp. 5-31, 2010.
15. D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, “Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study”, *in 21th IEEE International Conference on Software Maintenance (ICSM)*, pp.683-686, September 2005.
16. R. Hierons, “Testing from Partial Finite State Machines without Harmonised Traces,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1033 - 1043, 2017.
17. G. Juve, A. Chervenak, E. Deelman, et al. Characterizing and profiling scientific workflows. Future Generation Computer Systems, 2013, 29(3):682-692.
18. J. Li, W. Yao, Y. Zhang, H. Qian and J. Han, “Flexible and Fine-Grained Attribute-Based Data Storage in Cloud Computing,” *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 785-796, 2017

19. D. S. Linthicum, "Cloud Computing Changes Data Integration Forever: What's Needed Right Now," *IEEE Cloud Computing*, vol. 4, no. 3, pp. 50-53, 2017
20. B. Keshanchi, A. Souri and N. Navimipour, "An Improved Genetic Algorithm for Task Acheduling in the Cloud Environments Using the Priority Queues: Formal Verification, Simulation, and Statistical Testing," *Journal of Systems and Software*, vol. 124, pp. 1-21, 2017.
21. I. Ku, Y. Lu and M. Gerla, "Software-defined Mobile Cloud: Architecture, Services and Use Cases," *in The International Wireless Communications and Mobile Computing Conference (IWCMC 2014)*, pp.1-6, August , 2014.
22. Y. Wang, B. I. Jun, and K. Zhang, "A tool for tracing network data plane via SDN/OpenFlow," *Journal of Science China Information Sciences*, vol. 60, no. 2, pp. 022304:1-13, 2017
23. S. Wang, J. Zhai, H. Zhu and X. Wang, "Parallel Ordinal Decision Tree Algorithm and Its Implementation in Framework of MapReduce," *Machine Learning and Cybernetics*, Springer Berlin Heidelberg, pp. 241-251, 2014.
24. H. U. Yannan, W. Wang and X. Gong, "On the feasibility and efficacy of control traffic protection in software-defined networks," *Journal of Science China Information Sciences*, vol. 58, no. 12, pp. 1-19, 2015.
25. Q. Yan, F. R. Yu and Q. Gong. "Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS)Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges". *IEEE Communications Surveys Tutorials*, vol. 18, no.1, pp. 602-622, 2016.
26. T. Yen and C. Su. "An SDN-based Cloud Computing Architecture and Its Mathematical Model". *in International Conference on Information Science*. pp.1728-1731, 2014.

Liqiong Chen received his B.S. degree from Anhui University of Technology in 2004, and Ph.D.degree from East China University of Science and Technology (ECUST) in 2009. She is presently an associate professor of the Department of Computer Science and Information Engineering Shanghai Institute of Technology. Her research interests include formal methods for complex software systems.

Yunxiang Liu received his B.S. degree from Northeast Normal University in 1994, and Ph.D.degree from Jilin University in 2003. He is presently a professor of the Department of Computer Science and Information Engineering, Shanghai Institute of Technology. His research interests include artificial intelligence, computer software and theory, information fusion.

Guisheng Fan received his B.S. degree from Anhui University of Technology in 2003, M.S. degree from East China University of Science and Technology (ECUST) in 2006, and Ph.D. degree from East China University of Science and Technology in 2009, all in computer science. He is presently a research assistant of the Department of Computer Science and Engineering, East China University of Science and Technology. His research interests include formal methods for complex software systems, service oriented computing, and techniques for analysis of software architecture.