Adapting Intelligent Test Design For Cloud System

Behailu Getachew Wolde [#1] and Abiot Sinamo Boltana [*2]

[#1] *Carl von Ossietzky University of Oldenburg*
Department for Computer Science, Software Engineering
6111 Oldenburg, Germany
wolde@se.uni-oldenburg.de

[*2] *Mekelle University, Ethiopian Institute of Technology (EiT-M)*
School of Computing
231 Mekelle, Ethiopia
abiotsinamo35@gmail.com

***Abstract*—Cloud system flourishes with the rapid growth of ready-made digital transformation and evolution. Meantime, organizations have become more active towards cloud application and services. However, in such a significant system behavior of service function is dynamic and unpredictable. At the end-user side where there is limited knowledge, this service needs a new testing strategy to ensure quality of its expected requirements. The reasons are that the server-side knowledge remains hidden, and client-side test implementations and executions pose challenging classical testing methods. One way of fulfilling these quality requirements is to create a test design based on the behavior of provided service specifications. Thus, the need to adopt an intelligent test design through this specification is essential. It aims to show the importance of test design and validate the quality of the cloud using sample test cases generated from the specifications. The validation phase uses the four-phase test syntax based on two instances of mobility service running on NEMo mobility platform. Furthermore, the proposed model is indicated as future work to describe the necessity of an artificial intelligence approach to plan and adapt test cases iteratively to gain maximum test coverage.**

***Index Terms*—Test Design, Software Testing, Artificial Intelligence Testing, Cloud System**

I. Introduction

Test design is a technique to maintain and improve the quality assurance especially in software testing processes [14]. This technique works for re-engineering the legacy and traditional ways for enhancing the application productivity. The primary purpose of test design is to create optimal test cases for ensuring the quality of software products. However, regardless of this fact, many organizations execute their routine tasks using traditional ways with little effort to know their work efficiency and productivity as intended. The usual approach procedures are less likely to attain the quality test criteria as per user needs. One reason is that the traditional local environment is not capable of testing instantly as many test cases as possible. Besides, the studies indicated [1], [3] that testing in such an environment is not affordable as the nature of testing duration takes more time which needs an extra budget and testing infrastructures. Therefore, this issue needs a new testing environment as well as an optimal test design to ensure the growth and sophistication of digital transformation and evolution. The rise of this digital information enables organizations to gain momentum in their important business functionality for executing various tasks via Internet technology. As a result of this technology, they can access free to reuse ready-made services and resources anywhere with no geographical restrictions [1].

Recently, this fact associates with the rapid growth of cloud computing. A cloud computing is the best option to create new opportunities, technologies, and solution. For example, artificial intelligence (AI) technologies become essential to analyze big data using this computing. AI supports to generate test cases from formal unified modeling language (UML) (e.g., activity diagram) [7], [16] based on user behavior and requirements. However, the use of AI has given less attention and remained untouched such as using a cloud system in software testing.

Furthermore, the top players like Amazon web services [3] and Microsoft Azure [4] and other enterprise service providers such as Web Service Oxygen (WSO2) [8] execute many services but, the applicability of these services for the end users require test design which helps to ensure their behavior and functionality. In this regard, providing the quality of cloud becomes an issue of our day to day assignments. The aim is to reduce challenges imposing on quality of applications and services. The primary task is to introduce test design based on the behavior of requirements to validate the cloud application. This behavior is defined by four-phase test pattern [5], [12] based on scenarios of selected cases from NEMo mobility service specification which are running on NEMo mobility platform via WSO2 cloud infrastructure [8].

The rest of this paper is organized as follows: next section describes the foundation which includes cloud, test analysis and test design approach. In section III, NEMo mobility services as case study is illustrated. In illustrating this case UML activity diagram is used. Besides, four-phase test pattern based on GivenWhenThen syntax is presented. In section V, AI test automation which gives overview of the importance intelligent algorithms (AI),

and suggested proposed model. In section VI, Validation is conducted, and this includes test case specification, implementation, and execution. Conclusion is discussed at the end.

II. Foundation

This section covers a brief of cloud computing, test analysis and test design approach.

A. Cloud Computing

Cloud computing is defined as layering of fundamental components which include unique characteristics, cloud service and implementation models [9]. These unique features include abstraction, scalability, on-demand service provisioning, usage based utility, pooling resources and high accessibility. A brief of these features are as follows [1], [6]:

- underlying hardware and platform is abstracted and provisioned as a service.
- enabling a scalable and flexible implementation.
- offering on-demand service based on service level agreements.
- managing pay per use model without up-front commitment by cloud users.
- performing a shared pool of resources for multi-tenant environment.
- Being accessed over the Internet by diversified clients.

Cloud service models are usually differentiated as Software-as-a-Service (SaaS), i.e., web application such as online shopping and banking service, Platform-as-a-Service (PaaS), which allows customers to deploy their own applications, and Infrastructure-as-a-Service (IaaS), which provides, for example, processing power or storage. The implementation model helps to execute the application. This implementation is classified into four deployment models. These models are private, public, hybrid and community cloud. The private is owned by a specific company under more control environment, public is designed for public use by any users, hybrid combines both private and public, and community gives an exclusive use by a specific community of clients from stakeholders that have common concerns. Thus, cloud can offer many services for the end users using these deployment models.

These services are provided by the implementations somewhere on the internet which is based on a Service Level Agreement (SLA). SLA is a non-functional requirements that realize the application functionality through running on the infrastructure of the cloud. Realization of functionality and SLA can only be possible through creating an effective testing process and making feasible test experiments using minimum test cases. One of the test case process is test analysis. Test analysis is a precondition for test design to generate test cases.

B. Test Analysis

This section describes test analysis which is a preparation to generate relevant test conditions. The test condition is based on knowledge of test object (e.g., test methods of an application) that determines test design using test design techniques (e.g., boundary value analysis, equivalence partitioning, etc) to create test cases. The purpose of creating test cases are to realize the test coverage items (i.e., test items covered in the application) by exercising the test suites. Test coverage item is produced by the test design techniques [4] and eventually validated by the test system on the System Under Test (SUT), which produces a test result showing a verdict of ”pass”, ”fail”, and ”inconclusive”. These test results help to calculate the metrics of a test coverage such as functionality, efficiency, and reliability.

Test data is a sub-process of test analysis which is configured for test configuration to be used as precondition to map with the test system. This system is a test environment that includes templates for abstract test cases (ATC), test data, and test configuration [14]. A test data uses the data types which are sourced from requirement specification and/or web service description language. ATC has a nature of language and platform independent, which can be easily adapted to any platform specific project. Once the ATC is created, the standard test system can be used for adaptation, code and test case execution. Building this ATC requires a systematic approach which includes test design techniques.

C. Test Design Approach

Test design is part of testing process that enables to transform generic test objectives into specific test conditions. Test design contains set of test activities that can describe a test object from which the required test basis, test rules or test requirement are derived. Hence, like any software testing, testing of cloud application requires to have relevant test design so as to capture an optimal test cases. Test design primarily has the following two major benefits:

1) adapting optimization techniques to attain cost effective solution with minimum test cases. Example: the use of AI technologies help to adapt optimization of test cases [7], [16].
2) enhancing execution through running many test cases simultaneously based on capability of built-in cloud functionality. Built-in cloud functionality is a quality attribute model which includes functionality, performance, security, and etc [1], [6].

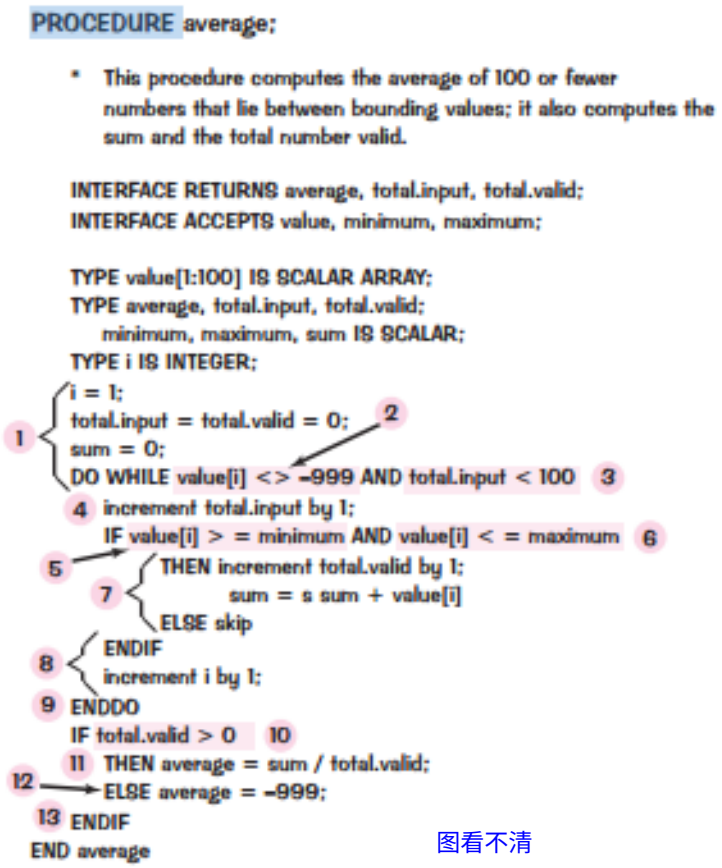
A cloud application can be ensured when an approach is assumed to consider these test design benefits. The efforts to achieve these benefits also enable software testers to have a new perspective in the foreign environment and help them to emulate the real time cloud for maximizing all test quality coverage. This study is also essential to show that the idea of test design is not limited only to non-distributed but also to cloud solution in a distributed environment. The basic issue is why test design is needed.

The common answers to this question can be associated with the following four key concepts [7], [15]: **(1)** To enhance leveraging the microservices (i.e., atomic services) optimally at the desired level that users use them. **(2)** To find out defect input pattern behavior when suspicious task happens at runtime. **(3)** To progressively adapt the outputs when data set for exercising becomes increasing. **(4)** To gain consistent real-time quality validation. For instance, a cloud leverages these key features from the AI technologies (e.g., genetic algorithm, simulated annealed, etc.) and the testing frameworks (e.g., JUnit, TestNG, JMeter, Assertion, etc.). These technologies have the ability to generate test cases. Then, test frameworks can be used to describe its observed or desired behavior at the runtime period during test execution.

However, cloud has a dynamic and complexity behavior that creates a barrier to impose on the pillar of its quality attribute models. Thereby, this makes testing not to be an easy task. So, test design has to be adaptable intelligently with the number of test cases exercised to capture the real time scenarios based on cloud behaviors. Here, the concern of test design is not primarily to compare the idea of manual with automated one, but to know the test design at which it depends on what knowledge patterns built in the process and ultimately to derive and produce test cases to test the cloud capability. Capability refers to the quality of cloud that enables to interact with the application and its components (i.e., service implementation). Specifically, test design tasks focus on the design of test cases to leverage this capability. This paper focuses on forwarding this idea to know how exercise NEMo mobility services (see *section III*). The scope of this paper intends only to show the importance of test design, but not to show the detailed aspects of various test design techniques.

Test design approach is deemed to build suffice test cases which are used to verify the adequacy of criteria in the specified test design [4]. Practically, exhaustive or complete test design is impossible. For elaboration of this testing principle, assume the software program which is stated for average computation in Pressman [11] (see Fig 1). Consider that the algorithm of this program is required to be executed in a cloud environment. This program has 100 lines of code. In a few basic parameters declaration, the program has two nested loops that run from 1 to 20 times each relies on the rules specified at value and input.

Within the interior loop, four if-then-else internal logic rules are defined. There are roughly 10^{14} possible paths in the provided algorithm specification. Here, path refers to a test case that can be introduced based on test design approach [4]. The number 1-13 circled indicates the independent paths where testing should be precisely induced. To demonstrate all paths in a given program, one considers that a magic test processor (”magic” because no such processor exists) can be developed for exhaustive testing. The processor can build a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause abnormal conditions in most implementation schedules. So, it is fair to assert that exhaustive testing is impossible for large and complex automating systems such as application running in or of a cloud environment .



图看不清

Fig. 1. Average Procedure Design (Algorithm) [4]

In other scenario, what ever effort is generated to attain optimal test design, *”testing cannot guarantee the absence of faults”* [4]. If this is true to happen, how to obtain subset of test cases from all range of test cases with a high probability to predict majority faults? The fundamental means to this question is to have test case design, which includes internal as well as external test logics to a SUT. Test case design approaches can be categorized from two perspectives:

1) Internal program logic, white box-testing : A software testing method in which the internal implementation of the White Box item being tested is known to the tester. Test design approaches include: Control flow Testing (e.g., condition testing, data flow testing, and loop testing), Path testing (e.g., Flow graph, independent program paths (i.e., cyclometric complexity), graph matrices).
2) Black box-testing : A software testing method in which the internal implementation of the item

being tested is not known to the tester. Test design approaches include: equivalence partitioning, boundary value analysis, cause effect graphing, orthogonal array testing, and model-based.

The idea of knowing these test logics help to decide the appropriate test design approach. This decision has an influence in finding the optimal test cases in SUT. These test cases have to be prepared to test both sides being used at the level of client and server-side environment. Test case design on the server side uses a standard technology such as specification and web service description language [13]. From this technology, abstract test cases (i.e., server side test design) can be generated, and then, test implementation and execution at client side can be instantiated using test driver on standard test frameworks and development environment. For dynamic adaptation of test design test automation and intelligent algorithms should be considered so that a consistent real-time verification and validation would be ensured. The process of generating test cases by AI approach will not be covered in this paper, but the proposed object model and its approach are introduced for further extension of this work.

III. Overview Of AI Test Automation

This section introduces the importance of AI test automation and proposed UML modeling.

A. Necessity Of AI

The world is becoming towards adapting intelligence agent in part of life such as anywhere at anytime from driver-less driving, how object-based message exchanges to use cell phones are controlled, and how big data analysis and monitoring are managed in a cloud, how complex and non-determinant tasks will be predicted and adapted in the future [15]. All these indicate the application of AI becomes essential in our existing scenarios. Since an execution of an application and its management are only enhanced with AI software by adapting themselves to meet the specified user behavior and interests. The most notable example of the cloud provider is Amazon.com that attempts to modify AI service based on the extensive behavior of users life [15]. Other providers such as Microsoft Azure and WSO2 cloud enterprise use the same functionality to maximize their service efficiency. So, it is a need to start improving the advantage as well as reducing bottlenecks of intelligent algorithms.

In software testing as noted in [4], [13], test automation is the application of specific software to control the execution of tests by comparison of actual results with predicted results. Test automation aids to automate repetitive but necessary activities in a formalized testing process already in place, or perform complementary testing that can be difficult to do manually. Once test automation is developed to test a given scenarios, it is often possible to reuse the test cases with minor refining for other similar tasks such as test cases for testing components of a cloud for distributed clients. However, the complex input variables to interact with cloud application by diversified clients make test design challenging. So, testing distributed application for various clients require a new approach that enable to integrate with one or more AI algorithms for achieving optimal test cases.

By doing this, the built-in nature of such algorithms have capability to support planning and diagnosing new test cases which are not used before for testing purpose [7], [16].

B. Proposed Object Modeling

The section describes the proposed solution using UML modeling. This modeling is a conceptual model which shows the proposed solution for test design in a cloud system. In Fig 2, each object (i.e., component, service, test design, etc.) in the model is represented to show the presence associations and relationships among each other. This conceptual model can be concretized based on the specific service that helps to initialize and transform into an intelligent test design. The service is defined by the service implementation (i.e., component). The test design uses a formal model of service such as the dynamic activity diagram (see section IV A).

怎么就是智能的测试设计了？

动态的活动图

This diagram has two essential model features which refer to a high-level model. One is the data flow and the other is object (control) flow. The former refers to the data which goes out from one node and goes in to the immediate next node. The latter controls the safety of the former to pass through it. These features also pose the validity of the model and are proved by model checker such as NuSMV [10]. This diagram can serve as an input to create many test cases. For instance, each node and edges of the activity model can be inputs in AI algorithms [16]. Test cases can be updated by these algorithms that help to predict and derive new test cases, which may not exist in the previous test bed. These algorithms are useful to attain optimal test cases through selecting by planning and prioritizing by diagnosing test cases iteratively. The new test cases can be derived, and the optimization process continues seamlessly to reduce the test cost for increasing and enhance test coverage for improving productivity respectively. Fig 2 explains a high level design which

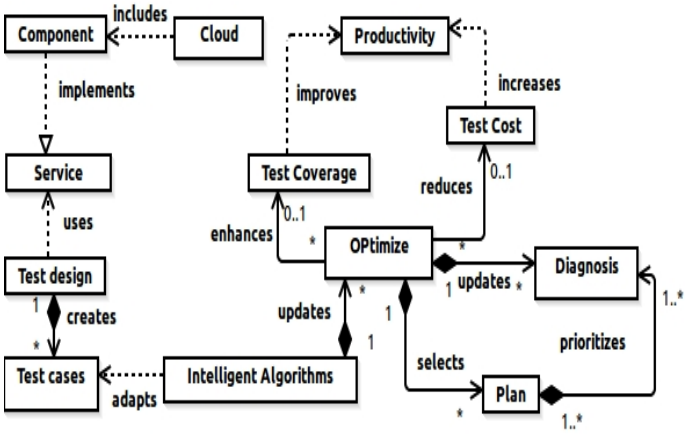


Fig. 2. Proposed Model For Test Design With AI-Approach

includes how to optimize the test cases in adaptive or dynamic manner through the AI approach. With adaptive approach, the change of parameter values are according to the input-output operations using search memory progress whereas in dynamic update the change of parameter values are performed without care for about memory search. Hence, AI approach supports to enhance the test design based on selection and prioritization algorithms. This solution attains two major quality metrics [15]:

1) Enhance test coverage
2) Reduce test costs

1. Enhance test coverage: Since testers are using minimum test cases using unlimited resources from the cloud and thus, the chance to execute all test cases are certain. The other reason is that cloud capability provides simultaneous or parallel execution and this enhances the test coverage of test item.

2. Reduce test costs: Test cost relates with the total time to complete all test case execution. With a cloud, the time schedule is strict. This means, for additional test time, additional resource can be consumed, and if that is the case the cost will be ”pay per use model”. By attaining these metrics, the productivity can be maximized with less costs for infrastructures for testing purpose.

IV. Case Study

This section describes the NEMo Mobility Platform with respect to NEMo In-Out sequence, selected NEMo test instance and four-phase test pattern.

A. NEMo In-Out Sequence

NEMo mobility platform primarily consists of four core REST mobility services to provide inter-modal routing which aims to achieve sustainable and flexible transportation based on demands to satisfy the customer. Fig 3 depicts the in and out activity which are represented using input and output line of UML activity diagram. The user or client requests the system from the NEMo mobility platform to get the best route for mobility service. *Orgin*

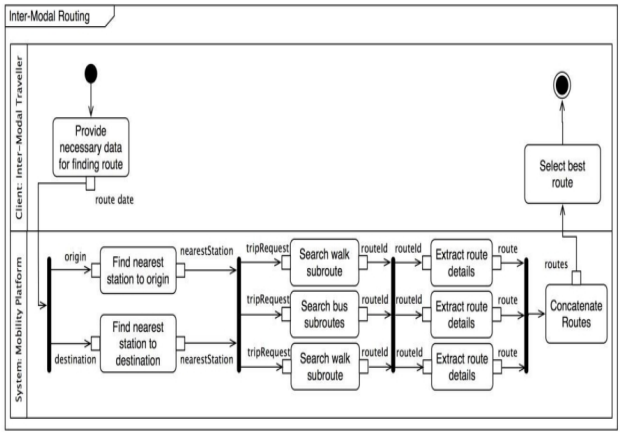


Fig. 3. NEMo Mobility Service Activity [8]

and *destination* are encoded to initiate the inter-modal routing. This initial data entry locates and receives the location id from the NEMo Mobility platform. This location id is an input to the next activity which is *find nearest stations* to origin and destination respectively.

find nearest stations verify location id of each mode (i.e., mode is a transportation means such as bike, bus, and walk) and *tripRequest*. A subroute for each respective mode is extracted as output. Each subroute produce a routeid which contains Google map id to identify the route of each capability mode. In the inter-routing mode more than one mode can be involved and each mode will have its own routeid. As a result, if there are three modes, three routeid are extracted and each routeid will form an independent route, i.e., route 1, route2 and route3. The final activity node is to concatenate these three routes and select the best route based on the time in which it consumes to complete the mobility. The time used for each route is compared and the smallest one is informed to the client. Each node and edge in the activity diagram also follows a path to determine the graph. This indicates that the use of AI can be easily adapted through embedding into this diagram for generating test cases. Luckily, majority of AI technologies work based on graph algorithm, e.g., Genetic algorithm [7], [16].

B. Selected NEMo Test Instance

The NEMo mobility service is implemented to be applicable and flexible to provide mobility for people in the rural areas [8]. This mobility service includes the services which are required in this work. In NEMO, mobility services have the signatures that include service data points such as names, descriptions, input and output parameters, and its data types. These data fields are used to implement the NEMo services. Two service instances are selected from mobility services. These instances are *RouteDetails* and *RouteConcatenator* (see Fig 3). The two selected instance services may show normal scenario without a procedure to test their quality features, and they might also show negative scenario in the case when they are exposed to a good test case.

The chosen services have basic elements such as *End point*, *Input request* and *Output response*. *End point* associates with each service interface. In the case of NEMo, services are implemented based on their REST interface identifier. Identifier can be a base URI plus resource. Base URI refers to the server or host, and resource refers to a specific base Path. *The Scheme=”http://baseURI[:port]/resource/”*, where *port* indicates a service in which the web server is configured. *Input request* contains test data and data types. For instance, *RouteConcatenator* test request receives the routes from *RouteDetails* as input parameters to determine the route. Each route contains a routeid of each stopover mode. *RouteDetails* test request contains *routeId* as parameter which uses a string type. *Output response* is the expected results which present JavaScript Object Notation (JSON) data. NEMo project uses JSON data interchange as data representation at the client side.

C. Four-Phase Test Pattern

This part helps to describe the scenarios based on GivenWhenThen behavior driven development (BDD) approach [12]. This approach underlines the specified user's data behavior on System Under Test (SUT). In [2], the four-phase test has four parts to test the SUT. Such parts are *fixture setup*, *exercise SUT*, *result verification* and *fixture teardown*.

fixture setup or Given: the pre-condition to test the SUT. It describes the state of world based on the specified scenario. *exercise SUT or When*: tester interacts to a SUT which describes its behavior. *result verification or Then*: this phase describes the changes for specified behavior. *fixture teardown*: at this stage, cleanup and reserve resources to return back to its previous state will be done. Based on this pattern, the scenario of selected NEMo test instances for *RouteDetails* and *RouteConcatenator* are stated. As it is noted below, these scenarios are adapted from Fig 3:

1) Scenario For *RouteDetails*

```
Feature:  route identifiers produce routes.
Scenario: executing testRouteDetails
        Given the value of routeid_1 is <routeid_1>
        And the value of routeid_2 is <routeid_2>
        And the value of routeid_3 is <routeid_3>
        When testRouteDetails is called
      Then the value  routeid_1
      And the value   routeid_2
      And the value   routeid_3 should be executed
Examples:
| (<routeid_1>)   |(<routeid_2>)   | (<routeid_3>)  |
|   route_1       |   route2       |   route3       |
```

Fig. 4. Given When Then Scenario For Route Details

2) Scenario For *RouteConcatenator*

```
Feature:  User gets the best route
Scenario: executing testRouteConcatenator
        Given the value of routeid_1 is <route_1>
        And the value of routeid_2 is <route_2>
        And the value of routeid_3  is <route_3>
        When   the value of route_1= route_1 (routeid_1)
      And the value   of route_2=route_2(routeid_2)
      And the value of route_3=route_3(routeid_3)  should be concatenated
      Then  one of the routes should be executed
Examples:
| (<routeid_1>)    | (<routeid_2>)   | (<routeid_3>)  |
| route_1          | route_2         | route_3        |
```

Fig. 5. Given When Then Scenario For Route Concatenator

The GivenWhenThen approach contains feature to be tested using the specified behavior in the scenario. In the scenario each test instance has inputs and outputs. For test instance *RouteDetails*, the inputs are routeid_1, routeid_2, and routeid_3 and the outputs are route_1, route_2, and route_3. In the inter-modal routing [8], a flexible and sustainable mobility mode can be serving to provide mobility service. If the capability works for three mode, then three routes can be extracted with their corresponding routeids. Accordingly, test instance *RouteConcatenator* receives these three routes to decide one best route for the user. Formalizing these rules can be difficult if there no proper specifications that meet these scenarios. This work extends the test specifications from work that is defined in [8].

V. Validation

The validation includes test case specification, implementation, execution and visualization.

A. Test Case Specification

Test case specification refers to the test plan which includes input request (test instance), test data, expected results, output responses and the test environment. The test case specifications are made at the level of server side from test case design. These test cases will be used for implementation at the client side so that testing the cloud can be conducted as intended. The given values of test instances are as follows:

1) Test instance **RouteDetails()** end point: "http://baseuri/VerticalPrototypeDummies/ services/ routedetails/RouteDetails/invoke"
   RouteDetails() input: "routeId_1":"05903d72-557f-4130-b009-dffe9055f33a"
   RouteDetails() input: "routeId_2":"33b9f7f6-efe5-429f-8bc8-32fb567b9eed"
2) Test instance **RouteConcatenator()** end point: "http://baseURI/VerticalPrototypeDummies/ services/routeconcatenator/ RouteConcatenator/ invoke"
   RouteConcatenator() input: "route_1", "Fuweg|53.14381,8.2214|53.14347,8.22104| 53.14352,8.22085|53.14375,8.21953| 53.14349,8.21454|53.14293,8.21368"
   RouteConcatenator() input: "route_2", "Radweg|53.14381,8.2214|53.14347,8.22104| 53.14352,8.22085|53.14349,8.21454| 53.14325,8.21313|53.14293,8.21368"
   RouteConcatenator() input: "route_3, "Fuweg|53.14381,8.2214| 53.14347,8.22104|53.14352,8.22085 |53.14375,8.21953| 53.14349,8.21454| 53.14293,8.21368"

TC#	Test Instance	Test Data	Expected	Output Response
TC-1	Route Details	route id_1	route_1	
TC-2	Route Details	route id_2	route_2	
TC-3	Route Concatenate	route_1, route_2, route_3	best route	

TABLE I
TEST CASE SPECIFICATION

B. Implementation

The test environment is conducted on NEMo Mobility platform in the cloud while test script writing and requesting is at the client side using Eclipse IDE and TestNG testing framework. Each test instance of the implementation has GivenWhenThen syntax as indicated in *Appendix A* and *B* respectively. The selected test instances *RouteDetails* and *RouteConcatenator* have a common global interface to allow tester to write the test driver such as input and output data interface by IOData implementation. This IOData object is provided to access the parameter values. The HttpResponse object uses this parameter value to update data by using endpoint (i.e., baseURI) and PUT HTTP method on NEMo mobility infrastructure, e.g., database. Partial view of this implementation is depicted in the *Appendix* section.

C. Execution and Visualization

The test execution and visualization is done at client side with TestNG annotation *@Test* for both *testRouteDetails()* and *testRouteConcatenator()*. Execution is done at the same time for functional tests. The interpretation of the results are as follows: Fig 6, the *TC-1, TC-2 and TC-3 test cases* were reported as *expected*. The verdict was PASSED status for all three test instances. This status indicates only functional correctness regardless of non-functional tests. *testRouteConcatenator* was successful to produce expected results based on the routeid_1 and routeid_2. The corresponding results were route_1 and route_2. These routes wired with pair of latitude and longitude values. The routes initial name indicate the place where the mobility begins, e.g., FUßweg and Radweg.

```
[RemoteTestNG] detected TestNG version 6.14.2
Parameter name: route_1, Value:
"Fußweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14375,8.21953|
53.14349,8.21454|53.14293,8.21368"
Parameter name: route_2, Value:
"Radweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14349,8.21454|
53.14325,8.21313|53.14293,8.21368"
Parameter name: route_3, Value:
"Fußweg|53.14381,8.2214|53.14347,8.22104|53.14352,8.22085|53.14375,8.21953|
53.14349,8.21454|53.14293,8.21368"

Request ==== > {"map":{"route_1":"Fußweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14375,8.21953|53.14349,8.21454|
53.14293,8.21368","route_2":"Radweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14349,8.21454|53.14325,8.21313|
53.14293,8.21368","route_3":"Fußweg|53.14381,8.2214|53.14347,8.22104|
53.14352,8.22085|53.14375,8.21953|53.14349,8.21454|53.14293,8.21368"}}

Response ===> {"map":{"concatenatedRoute":"{\"route_3\":{\"coordinates\":
[{\"lng\":8.2214,\"lat\":53.14381},{\"lng\":8.22104,\"lat\":53.14347},
{\"lng\":8.22085,\"lat\":53.14352},{\"lng\":8.21953,\"lat\":53.14375},
{\"lng\":8.21454,\"lat\":53.14349},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Fußweg\"},\"route_2\":
{\"coordinates\":[{\"lng\":8.2214,\"lat\":53.14381},
{\"lng\":8.22104,\"lat\":53.14347},{\"lng\":8.22085,\"lat\":53.14352},
{\"lng\":8.21454,\"lat\":53.14349},{\"lng\":8.21313,\"lat\":53.14325},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Radweg\"},\"route_1\":
{\"coordinates\":[{\"lng\":8.2214,\"lat\":53.14381},
{\"lng\":8.22104,\"lat\":53.14347},{\"lng\":8.22085,\"lat\":53.14352},
{\"lng\":8.21953,\"lat\":53.14375},{\"lng\":8.21454,\"lat\":53.14349},
{\"lng\":8.21368,\"lat\":53.14293}],\"transportMode\":\"Fußweg\"}}"}}

Parameter name: routeId_1, Value:
05903d72-557f-4130-b009-dffe9055f33a
Request ==== > {"map":{"routeId_1":"05903d72-557f-4130-b009-dffe9055f33a"}}

Response ===> {"map":{"route_1":"\Fußweg+53.14716-8.18845+53.14832-
8.18265+53.14832-8.18265+53.14641-8.18666+53.14641-8.18666+53.14945-
8.19073+53.14945-8.19073+53.14867-8.1946+53.14867-8.1946+53.14763-
8.19823+53.14763-8.19823+53.14744-8.19827"}}

Parameter name: routeId_2, Value:
33b9f7f6-efe5-429f-8bc8-32fb567b9eed
Request ==== > {"map":{"routeId_2":"33b9f7f6-efe5-429f-8bc8-32fb567b9eed"}}

Response ===> {"map":{"route_2":"\Radweg+53.14716-8.18845+53.14832-
8.18265+53.14832-8.18265+53.14347-8.19292+53.14347-8.19292+53.14359-
8.19304+53.14359-8.19304+53.14353-8.19312+53.14353-8.19312+53.14423-
8.19346+53.14423-8.19346+53.14425-8.19889+53.14425-8.19889+53.14744-8.19827"}}
PASSED: testRouteConcatenator
PASSED: testRouteDetails
PASSED: testRouteDetails2
===============================================
    Default test
    Tests run: 3, Failures: 0, Skips: 0
===============================================
===============================================
Default suite
Total tests run: 3, Failures: 0, Skips: 0
===============================================
```

Fig. 6. Test Result of Console Visualization

VI. Conclusion

The overall intention of test design is to create optimal test cases based on the requirements and eventually to get the quality of product. This initial effort helps to enhance both test coverage and test cost effectiveness of resultant output in many perspectives. The test implementation motivates towards exercising more test cases to attain maximum test coverage. In a large system like a cloud, manual preparation of test cases are difficult and also, require optimal test case specification from the server side. So, a test design has to be automated with adaptive approach like AI algorithms that plan and diagnose test cases iteratively by emulating and predicting a cloud environment. For instance, NEMo mobility services have dynamic variables that require AI approach to get sufficient knowledge of mobility during the real-time events. So, researches on test design based on AI optimization techniques are needed to consider more than ever before. Furthermore, optimal test cases should be done by cloud team collaboration which enhances the quality of test cases to reuse for other similar related testing functionalities. Therefore, the proposed model shows an alternative way to generate good test cases by using the built-in capability of AI-algorithms which can realize both explicit and implicit behavior of a specified service.

Acknowledgment

A major part of the research reported in this paper is carried out by a scholarship granted from Ethiopian Engineering Capacity Building Program (EECBP) in collaboration with DAAD. We are highly indebted and credited by the gracious help from University of Oldenburg, Software Engineering Group for their constant support while testing sample instances of mobility services on NEMo mobility cloud platform.

References

[1] Kees Blokland, Jeroen Mengerink, and Martin Pol. *Testing cloud services: how to test SaaS, PaaS & IaaS*. Rocky Nook, Inc., 2013.

[2] Four-Phase Test, http://xunitpatterns.com/four%20phase%20test.gif. feb 09, 2011 [dec. 04, 2018], 2011.

[3] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal*, 1(1):9–23, 2011.

[4] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[5] Javier J Gutiérrez, Isabel Ramos, Manuel Mejías, Carlos Arévalo, Juan M Sánchez-Begines, and DAvid Lizcano. Modelling gherkin scenarios using uml. 2017.

[6] CN Höfer and Georgios Karagiannis. Cloud computing services: taxonomy and comparison. *Journal of Internet Services and Applications*, 2(2):81–94, 2011.

[7] Rijwan Khan, Mohd Amjad, and Dilkeshwar Pandey. Automated test case generation using nature inspired meta heuristics-genetic algorithm: A review paper. *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, 3(11), 2014.

[8] Dilshodbek Kuryazov, Andreas Winter, and Alexander Sandau. Sustainable software architecture for nemo mobility platform. 2018.

[9] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. *National institute of standards and technology*, 53(6):50, 2009.

[10] Faiz UL Muram, Huy Tran, and Uwe Zdun. Automated mapping of uml activity diagrams to formal specifications for supporting containment checking. *arXiv preprint arXiv:1404.0852*, 2014.

[11] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[12] Thiago Rocha Silva. Definition of a behavior-driven model for requirements specification and testing of interactive systems. In *Requirements Engineering Conference (RE), 2016 IEEE 24th International*, pages 444–449. IEEE, 2016.

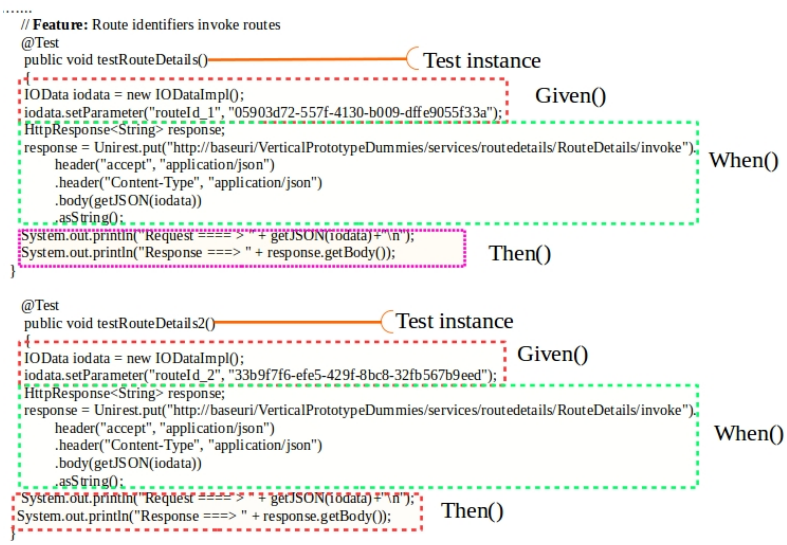
[13] Harry M Sneed and Chris Verhoef. Measuring test coverage of soa services. In *2015 IEEE 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, pages 59–66. IEEE, 2015.

[14] ETSI Standard. Methods for testing and specification (mts); the testing and test control notation version 3; part 6: Ttcn-3 control interface (tci). 2005.

[15] Peter Stone, Rodney Brooks, Erik Brynjolfsson, Ryan Calo, Oren Etzioni, Greg Hager, Julia Hirschberg, Shivaram Kalyanakrishnan, Ece Kamar, Sarit Kraus, et al. Artificial intelligence and life in 2030. *One Hundred Year Study on Artificial Intelligence: Report of the 2015-2016 Study Panel*, 2016.

[16] V Mary Sumalatha and Dr GSVP Raju. An model based test case generation technique using genetic algorithms. *The International Journal of Computer Science and Application*, 1(9), 2012.

Appendix A
Test Implementation-RouteDetails() and RouteDetails2()



Appendix B
Test Implementation-RouteConcatenator()