

Behavior-Driven Quality First Agile Testing for Cloud Service

Behailu Getachew Wolde, Abiot Sinamo Boltana

School of Computing, Ethiopian Institute of Technology (EiT-M), Mekelle University, Mekelle, Ethiopia

Email address:

behailu.getachew@mu.edu.et (B. G. Wolde), abioticsinamo35@gmail.com (A. S. Boltana)

To cite this article:

Behailu Getachew Wolde, Abiot Sinamo Boltana. Behavior-Driven Quality First Agile Testing for Cloud Service. *Software Engineering*. Vol. 9, No. 1, 2021, pp. 9-35. doi: 10.11648/j.se.20210901.12

Received: April 25, 2021; **Accepted:** June 24, 2021; **Published:** June 29, 2021

Abstract: In a recent testing software system, with cloud implementation a cloud service abstraction allows providing high-level ubiquitous language (UL) compositions through behavior-driven step-wise agile cycles. By adapting this UL the instantiated behaviors simplify the demand in software quality through mechanisms to cope with complex digital transformation and evolution. However, in such a context, testing becomes challenging quality engineering. As a result, it poses a threat to the services on a cloud as the access to its source codes rely on these abstractions. The aim is to introduce a way by strictly focusing on a black box approach. One way is to realize the client-side continuous Quality First (QF)-Test behavior-driven development (BDD). On this point, a meta-model helps to transform the RESTful cloud specification through domain specific language (DSL) while accommodating a low-level details on a test coverage report. By using features from the user stories the Gherkin BDD styles enable the meta-model which creates the instances of DSL to implement the runnable test steps on a cucumber framework. Each step is designed in a GraphWalker through modeling a context finite machine via a model visual editor, and generates the dependency test model paths. As an evaluation, the QF-Test executes the required steps given by data-driven elements for creating run-log trace analysis. As a comparison, the Jenkins framework is configured to build the QF-Test node of test suites for generating the behavior-driven and continuous integration test report. Moreover, the steps with the keywords are automated to verify the GraphWalker test cases through traversing the paths generated. As a case application, a sample REST API mobility service instance is considered.

Keywords: Cloud Service, REST API, Domain Specific Language, QF-Test BDD, GraphWalker, Model Path Validation

1. Introduction

This work orders the concept of Behavior-driven Development (BDD) which is defined to support the model path generation as in [1]. "This paper is an extension of work originally presented in conference name [1]." However, the results indicated in the conference demonstration dose not show the capability how to design a model path with concrete tests and quality first approach to fully validate the model specified in the BDD formation. In such a context, such extended paper aims to apply functional realizations with well-defined software engineering modeling techniques. The results obtained from these realizations are the predominant contributions to attest the intended behavior-driven model path approach in a cloud computing. The paper also extends a few examples from previous proceedings published in [1, 42]. In [42], the authors describe a feature model oriented

combinatorial testing approach through which it shows the possibility of enabling an exhaustive test case generation. This combinatorial approach shows the importance of array coverage measurement over data-driven testing with parameters taken from cloud based service. In [1], the authors develop an approach to define the possibility of equivalency data formats through re-engineering from one form (e.g., Java Architecture based extensible markup language-REST Service (JAX-RS)) to another (e.g., JavaScript Object Notation (JSON)). This re-engineering idea considers model constraint checking based on the idea of uni-direction graph walker to define a finite state machine (FSM) behavior-driven development (BDD). Here, the researchers mainly focus how model-driven engineering (MDE) supports to simplify the challenges of model inconsistency and differences by using meta-model transformation and synchronization into consistent instances of domain specific language (DSL). For this, it

defines a BDD from this language to instantiate test runner, and generates graph walker model to realize a comparable test coverage report over software quality frameworks such as a QF-Test industry software GmbH [32, 33] and Jenkins continuous integration (CI) [7, 14]. The rest of this paper is organized as follows: In order to understand the reasons why this study is important, in section 2, a brief of this motivation and requirements are explained. Section 3 describes the fundamental concepts which is a brief of cloud model. Model-driven engineering: meta-model, models and DSL instance, and BDD: cucumber BDD framework, and gherkin syntax. In section 4 the tools related to approaches are explained. Section 5 contains an approach that bases on the conceptual idea drawn in Figure 1 to generate a domain specific language (DSL). Section 6 explains application and data in brief on NEMO service with its minimal real time route assumptions to make use of parameters and operation as a query for finding the best route available for modes. Section 7 underlines the evaluation process and comparisons results on the specified NEMO sample instance, i.e., SUT. Further, a step by step method implementation with TestNG testing framework allows for model checking through assertions and GraphWalker algorithms. Following, the conclusion gives lessons learned.

2. Motivations and Requirements

Elaborating the following key points are vital: ways for testing cloud service, *contributions, stakeholders and requirements*, and *conceptual idea to enable test coverage*.

2.1. Ways for Testing Cloud Service

Cloud service specification is a software product document which expresses all high level features with language and platform independent like an eXtensible Markup Language (XML) or REpresentational State Transfer (REST)-based service composition [43]. In practice, one notable way is a request via service specification. This practice follows a model-based testing (MBT) based on high level functional requirements. In a software development process, these requirements have to be semantically transformed into the software models to define a small specific language. This transformation enables to preserve models' consistency and completeness [23]. However, for this to be ensured a manual test design is challenging as the states (or traversing inputs) to be asserted or checked against expected outputs are increasing on the given DSL. In fact, this issue has been partly demonstrated with examples through parameterized array combinatorial coverage with an automating black box testing strategy [42]. In addition, an effort of model consistency checking to enable the behaviors using a MBT tool is elaborated and corrected while a test path generation is traversed from a digraph test model (e.g., GraphWalker behavior-driven modeling [13]) [1].

Another way is a request via service interface. This request has capability to instrument basic test case definition such as precondition, inputs, post-condition and outputs [39]. For instance, it helps to enable a query on the specified REST

Application Program Interface (API). For this, a REST client side support tool is useful to parse the input query of the interface request (or API call) which discloses the outputs in terms of application JSON or XML message format exchanges. Usually, this message representation contains array and/or object responses. An object refers to the methods with variables associated to the System under Test (SUT). The variables take the data types as arguments defined in an XML or a JSON schema, and the methods take the parameters, too. With this respect REST API combines the above mentioned two ways to examine the abstractions which bind the hinge between business process and cloud model. Through this API approach a component and integration take 50% coverage share each respectively in software testing hierarchy. However, if we consider each module as a traditional testing does, it takes 100% coverage analysis. This testing becomes hard for a regular testing as the source codes such as implementation and infrastructure remain hidden to the test engineers. Thus, testing to conduct as usual imposes challenging on quality engineering. As a way out, the test engineers require a new alternative approach that allows them using high level API specifications as a testing strategy. The main purpose is to introduce QF-Test BDD in a MDE through extension that supports QF-Test plugins for Jenkins based on GraphWalker MBT for validating cloud based service.

2.2. Contributions

In order to implement this BDD continuous approach a test engineer uses test cases generated during behavior-oriented functional realization using a MDE. In MDE, one way of this realization becomes true via domain specific modeling language like a tooling support with a cucumber BDD framework [14, 37]. This framework eases the generation of automated tests through keywords combined with scenarios having test steps as classes which create a meta-model through Gherkin BDD styles. Then, the rule sets are developed to build FSM based on Graph-Walker principles. This FSM model contains a data structure which consists of state and edge for representing the meta-model in a model visual editor based on GraphWalker MBT. Furthermore, in [32], an industrial quality first software (e.g., QF-Test) capture and replay with a continuous integration build tool (e.g., Jenkins) use the test suites which compose behavior-driven scenarios with multiple test steps. The purpose of doing this is to generate the run-log file in a graphical hierarchy as well as to ease the trace out missing exception analysis which is not visible at the time of QF-Test capture and replay execution cycle. The QF-Test run-log generated report is further disclosed the hidden knowledge with Jenkins build configuration. Thus, the quality of product will be enhanced through only a careful design of BDD generation with standard automation test frameworks. In practice, the application is validated while the run-time low-level implementation presents itself to disclose the test coverage report on both QF-Test and Jenkins console output preview box. This run-time result helps to show the comparison reports with same test scenarios for both QF-Test graphical

user interface (GUI) framework environment and Jenkins QF-Test continuous delivery build analysis. In such a context, the paper includes running examples of high level scenarios over little programming skills among stakeholders, for instance, business owners and software developers.

2.3. Stakeholders and Requirements

The stakeholders to take part for tooling and delivering findings are:

- 1) *Meta Tool as Developer*: a Meta language transformation and synchronization through a model consistent checking based on UML re-engineering principles are considered in [4, 22, 23, 28].
- 2) *Gherkin Model Creator as Business Analyst*: an ubiquitous language emanated from the user story which enables writing test codes on cucumber for BDD to run a Gherkin feature definition [8, 9, 27, 31, 37, 38].
- 3) *Test Path Generation as Keyword-driven Graph Walker*: this walker uses the states as nodes and edges as forwarding inputs which form a uni-direction digraph to generate paths for validating a test model. This model represents a FSM from which a path generation is made to realize the Graph-Walker keyword-driven path verification analysis.
- 4) *Test Step By Step Implementer (or Test Cases) as Test Engineer*: the implementation of a node test suites dependency hierarchy which gives a coverage equivalent to an exhaustive combination or loops of independent executable test cases through configuring an automatic error handling within quality test (e.g., QF-Test [32]) and continuous build tool (e.g., Jenkins [2]) frameworks.

This work embraces mainly the requirements to:

- 1) Take cloud based REST API specification (e.g., NEMO REST Route Plan System [25, 34]) to generate some plain text structured data (e.g., JSON) and transform it to some object data format. Such an object has its own type and predefined properties order class, but it has no behavior or test steps.
- 2) Take the user stories to derive the behavior features and describe the scenario to implement the test steps based on Gherkin meta model. Each step definition in Gherkin syntax forms keywords to create GraphWalker test model.
- 3) Take actions for each test step within the scenario to generate coverage analysis.

2.4. Conceptual Idea to Enable Test Coverage

Below a text UML layout are drawn to define the high level conceptual idea which includes the main requirement nodes with core artifacts and approaches like a CloudModel, MDE, BDD and TestFrameworks (see Figure 1). This idea helps to enable test coverage analysis through MDE on the cloud model. MDE is a useful way to embed the model principles that bind a model based testing (MBT) tool. One of this tool is a GraphWalker MBT which models the finite state machine (FSM). This FSM enables the BDD which

defines the DSL styles, and codes the test step definition with gherkin syntax. The BDD code execution generates a test coverage report for making a decision on the quality of requirements over a specification model, that is, REST API. Each node is wired with standard symbols and notations which are given in Table 1.

Table 1. Notation Types with Symbols.

Types	Symbols [29]
Association	--, <-->, -->
Composition	--*
Aggregation	--o
Relationship	0.., 1..*
Generalization	< --

Through this high-level idea, we further understand the related concepts, approaches, application, and results.

```

1  "@startuml name"
2  node CloudModel{
3    RESTAPI "0..*" -- "1..*" application
4    AppServer.(RESTAPI,application)
5  }
6  node MDE{
7    RESTAPI--|..GraphWalker
8    GraphWalker -- MBT
9    GraphWalker --* FSM
10 }
11 node BDD{
12   FSM o-- DSLModel
13   DSLModel *-- cucumberMetaBDD
14   GherkinSyntax --o cucumberMetaBDD
15   GherkinSyntax --* TestStepDefinition
16   GherkinSyntax *-- Keywords
17 }
18 node TestFrameworks{
19   Keywords --> QFTest
20   QFTest --> QFTestCoverageReport
21   QFTest --o Jenkins
22   Jenkins --> JenkinsCoverageReport
23   QFTestCoverageReport "0..*" -- "1..*" JenkinsCoverageReport
24   (QFTestCoverageReport, JenkinsCoverageReport) .ComparedTestReport
25   ComparedTestReport --> TestResultInterpretation
26 }
27 @enduml

```

Figure 1. Conceptual Idea to Define Textual Plain model for high level abstractions of CloudModel, MDE, BDD, and Test Frameworks for Generating Comparable Test Coverage Report with its Test Result Interpretations.

3. Fundamental Concepts

This section is written to cover topics like a cloud model, Model-Driven Engineering (MDE) which explains meta-model, model transformation, model Visualization, and model equivalence for relating BDD to JSON and object serialization.

3.1. Cloud Model

Cloud model is defined as layering of fundamental components which associate with unique characteristics, cloud service and implementation models [29]. It is characterized with uniqueness to describe features essentially abstraction, scalability, on-demand service provisioning, usage based utility, pooling resources and high accessibility. Typical examples of such features with illustrations are included in [5, 15]. To mention a few of these: (a) underlying hardware and platform is abstracted and provisioned as a service. (b) Enabling a scalable and flexible implementation.

(c) Offering on-demand service based on service level agreements. (c) Managing pay per use model without upfront commitment by cloud users. (d) Performing a shared pool of resources for multi-tenant environment. (e) Being accessed over the Internet by diversified clients.

Cloud service models are available remotely and usually differentiated as Software-as-a-Service (SaaS), i.e., web application like services such as NEMo Mobility project, 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 with application server (AppServer). The implementation model helps to execute the application over AppServer, and 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, a cloud offers many ready-made cloud services to fulfill the demands for end users using capabilities of these deployment model options. Cloud services are provided by the implementations somewhere on the internet which is based on a Service Level Agreement (SLA). This SLA is a non-functional requirement that realize the application functionality through running on the infrastructure of the cloud. Realization of functionality with SLA is possible through behavior model-driven engineering which enables a meta-model to have multiple instances of domain specific languages (DSLs). Accordingly, a meta-model is transformed to a model for enabling the test model and translated into keywords based on BDD with ease configuration of settings on relevant frameworks. In this essence, MDE supports abstraction of high level descriptions for enabling testing of cloud services regardless of geographical locations. A highlight of this idea is as follows.

3.2. Model-Driven Engineering

Model driven engineering (MDE) becomes a fundamental paradigm that has capability to simplify the complex specifications in software engineering [21]. In a MDE, the use of models is the key artifacts of software process at all phases from analysis, design, and implementation to testing and vice versa. The most popular approach to the MDE is the Model Driven Architecture (MDA) as defined in Object Management Group (OMG) [23]. Specifications are modeled to simplify complexity, create its instances at a platform independent level and reuse through transformation tool to (many like a java, ruby, .net, etc.) platform specific implementations. For example, a model driven web services like a REST API is the composed specification of MDE to the domain of RESSTful web development where it may be useful because of dynamic evolution of web technologies and platforms [18]. Using these technologies, for instance, in ensuring consistent management, checking of MDA pattern to the REST web application at all levels of development

process from analysis to the generated implementation with model transformations is important.

Meta-models are explicitly described for conforming to the platform specific implementation with dynamic REST API. By doing this, explicit transformations enable an executable code generation for a broad range of behavior oriented technologies. Thus, during transformations, if two models are consistent, then the states (or node styles) and its existing forwarding in and out dependency events (or edge styles) between them are free from contradictions [26]. Co-existing various models to represent the same system services like a NEMo cloud based mobility route facilities in software project help to realize the behaviors from high to low and vice versa. By using these behaviors test engineers devise a systematic way to create instances of high level feature for executing low level service operations and parameters in a MDE. One of this ways is a BDD approach through re-engineering principle. In a software project, the models exist in different forms which usually associate with software levels (i.e., specification, implementation and testing) to represent a software development life cycle (see Figure 2). Each level maps with platform independent model (PIM), which contains a high level abstraction at conceptual idea, and platform specific model (PSM), which allows a DSL implementation embracing the business logic. PIM is a domain driven model that represents a specification like an API call to enable into different co-existing data format such as JSON, XML and etc.

This specification has to use a meta tool to transform into consistent DSL model which orders a behavior-driven design such as a cucumber BDD test definition in Java language [14]. In enabling this definition, a gherkin meta model tool uses the features that consist of scenarios with test steps as syntax styles to create a specific model instance. These styles hinge test suites model between the feature model and the test libraries within PSM framework (e.g., cucumber scripts in java model). Feature model contains behavior features from the BDD design to describe user stories for enabling the relevant functional units and executable classes, i.e., scenarios and test steps (see Figure 7). In PSM, structural features are generated indirectly through local environment either exemplary eclipse modeling (see Figure 9) or QF-Test industry certification framework (see Figure 15). This environment has capable to utilize the remote NEMo cloud mobility platform and infrastructure by using specific language implementation which composes a java model, test steps definition and executable cucumber tests (see Figure 8). A meta-model is realized as enumerated functional operations for defining the behaviors of system interaction with actions for each test step in the scenario [12]. In other words, meta models offer an instance of one or more DSLs through behavior-driven agile software development. These meta-model constraints are well-formedness through an UML (Unified Modeling Language) model, with inclusion of semantically equivalent UML notation and are checked by transformations.

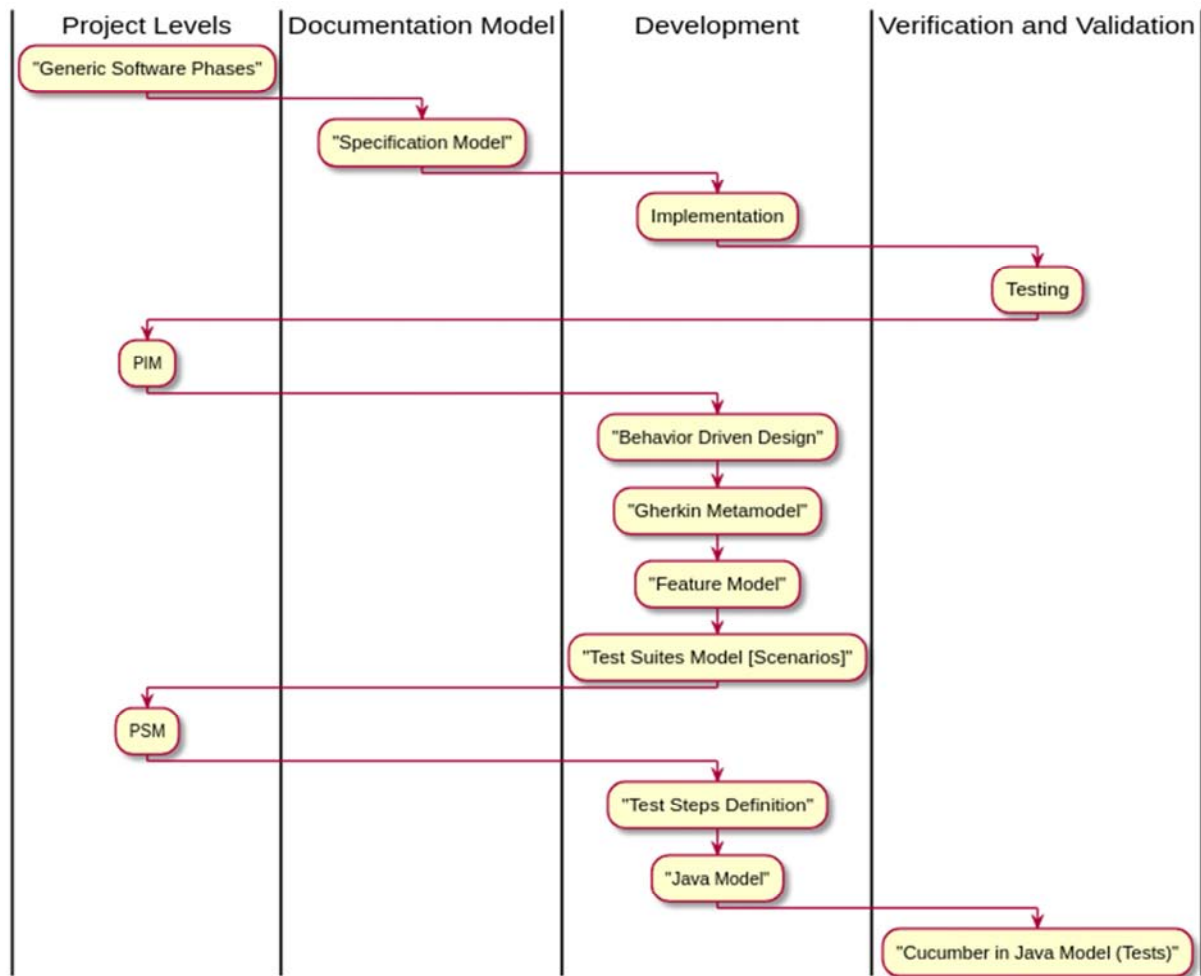


Figure 2. Different Domain Models Co-exist in Software Project.

Model transformation engine implements the formal evolution of analysis and design models [22]. The transformation to platform DSL model from the meta-model is decomposed along the concerns of services to cope with technology changes into a fine-grained way. The resulting models are serialized to code by means of serialization frameworks. For example, an object API specifies a specific object exchange format that a support code uses locally within an application, while an object exchange protocol expresses a way to transfer the same kind of information in a message sent to a remote system. When a message is exchanged via a protocol between services (e.g., classes) that are not present in Java, the classes that cannot be represented in Java object in a programming language are corrected. Such model correctness enhances the model quality as it supports a meta tooling to have a bidirectional or directional incremental consistency through model transformation and synchronization [23]. In Figure 3, a meta modeling tool receives or sends a model input like a JSON which is retrieved from the cloud to create meta model to be instantiated into a set of DSL model. This meta program (e.g., Meta Programming System (MPS) or Xtext DSL language) performs key tasks such as transformation and

synchronization. During this process, the tool conforms to the model consistency through its incremental algorithms with uni or bi-directional correctness between the current and previous model functionality which avoid the contradictions due to differences available in them [10, 23]. In version control system, the change in state due to deletion and update of co- co-existing software product bring a model difference representation that arises quality issues such as portability and maintainability.

Thus, the difficult to maintain this variation is true in the coexisting multiple domain models unless there is a testing strategy that can bug the missing aligned exception(s) to match with the idea around consistency management for supporting the software developer. In such case, this transformation algorithm will make a solution through reducing and mapping the rules to be corrected. The current output which passes using such process will be consistent with the expectation of target groups involved in accessing the service. This means, a model specification exists in JSON which is converted to another equivalent model form, a Java model with serialization. For this, an API message is sent using HTTP methods like a GET method for receiving a JSON model response which can be used as an input to

serialize and develop a Java model in an object-oriented design [19]. Regarding this co-existing consistency, we consider related approaches given API call to retrieve the PIM that helps to define MDE in the context of a MDA for realizing PSM (i.e., specific source code generation) and test coverage analysis. With this, the two typical approaches are vital for the purpose of this realization. The first approach creates a model serialization from one model form to another through JSONSchema2POJO [20]. A precondition to initialize this serialization process requires a generated input JSONSchema model. The second approach is to use known model consistent template such as meta programming system (MPS) that has capable to transform and synchronize (examples: Jet-brains MPS plugin) to meet the formal methods and software development in a MDE.) [6, 23, 28, 41].

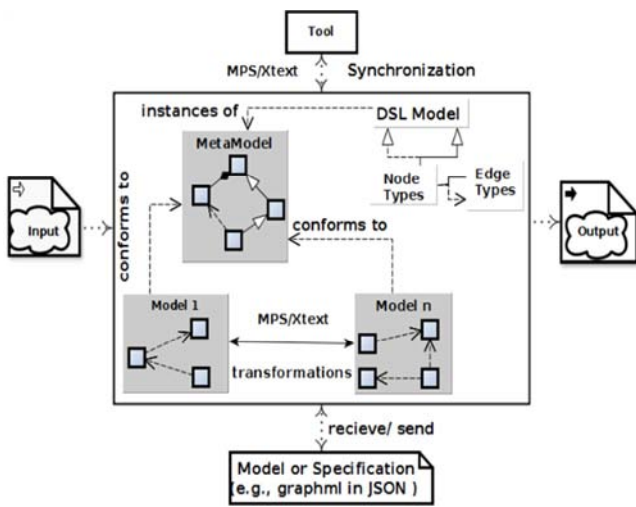


Figure 3. Meta-model to Model Transformation and Synchronization for Tool Creation to Enable DSL Instances such as using Jet-brains Meta programming System (MPS) or Xtext plugin (adapted from [28]).

In Figure 4, after JSONSchema model serialization, each `@JsonPropertyOrder` annotation takes the nodes as classifier as well as the edges as relationship to apply the behaviors interactions for the specified service. These behaviors take the NEMO CustomerService API specification example to read and generate JSON message such as using a postman REST client tool (hint: Postman is very popular client side REST service for extracting Rest API message from web AppServer:). Then, this message is used as an input in reading the JSON with `in.readValue (...)` for loading each JSON property order class through implementing `Serializable` and `Parcelable` interfaces. Once reading each property class in the JSON message is completed, the `@Override` annotation uses the method `writeToParcel (...)` with parameters `des` to write the JSON into Object and flags to check the binary 0 as false and 1 as true Boolean expression values. Such a conversion process is similar with the object-oriented programming (OOP) principles like a mutable and immutable functionality. In an OOP, the mutable refers to the use of setters and getters methods, which is writing (or updating) and reading (or retrieving) values to and from memory

respectively. The immutable refers to a return value which has already written to the memory regardless of the change in that state. Such a typical scenario exists when a class node has a definition of final scope without setters' method. These functionality operations in the OOP have analogy with serialization and de-serialization concept. For example, the NEMO mobility service is analogy with a Family hierarchy containment Family, Parent, mother and father, son (or Male) and daughter (or Female) in practice. This hierarchy contains Family meta-metamodel, Person object meta-model and Parent model which create instances of its own classifier and relationship to make realization of services (or abstraction of components) at the level of client side.

In similar fashion, in NEMO project layout, NEMO mobility platform as Cloud meta-meta model, NEMO mobility services as compositional meta-model, which enables list of enumerated specific NEMO project API to access operations like a *RouteDetails*, *TripRoutes*, *CustomerService* and etc. This implies, a meta-meta-model uses "is-a" generalization relationship to map meta-model instances, meta-model uses "is-part" composition relationship to create model object instances, and model forms an aggregation relationship to a system. A model (e.g., JSON) composes complex meta objects and arrays which should be transformed to create instances of DSL for describing general programming language (GPL) like a Java model. This GPL includes classifiers as states which should be verified and edges as relationship to move forward input and output events in a MDE like a finite state machine (FSM). Such FSM model draws a graph with the keywords (e.g., classes) using Graph-Walker principles on visual editor tool in Figure 13, and further verification of generated test sequences enables a complete coverage over the generated test path model in Figure 18. The running examples for model transformation and visualization processes are explained below:

- 1) *Model Conversion with JSONSchema2POJO*: this model conversion method is an approach to create coexisting different models with no contradiction based on OMG 2.0. One of these approaches is a JSONSchema2POJO which uses Jackson ObjectMapper to read the JSON generated from REST Client Service (e.g., Postman [3]) and to write into Object with JSON serialization (i.e., marshaling) and deserialization (i.e., unmarshaling) algorithms (see Figure 4). The typical use case for JSONSchema is to validate JSON message and API responses after an HTTP get method response is given to the Rest Client Service. In Linux, using the terminal command, *syntax*:

```
$ sudo /usr/bin/node /usr/local/bin/json-schema-generator -
-file jsonfilepath. /usr/bin/node path directory computes the
behaviors and validates the JSON file --file jsonfilepath to
create the output JSONSchema using /usr/local/bin/json-
schema-generator.
```

- 2) This output is used as an input for JSONSchema2POJO online tool to generate java source code service.
- 3) DSL and Language Workbenches: to design a DSL, use

of standard language workbenches are important. In presenting this DSL, an appropriate integrated development environment (IDE) supports JetBrains MPS or Eclipse Xtext plugins to enable custom extension on its DSL template for source code generation [16]. A DSL is a programming language that promotes the degree of abstraction with a specified solution that defines a template consisting of *concepts*, *properties*, *children*, *references*, and *concept links* from a specific problem domain such as descriptions for ubiquitous language (see Figure 6).

- Concepts*: to define classifier node type such as concrete class, interface, exception, annotation, etc.
- properties*: to define fields with data types such as a field "id" with a "String" class data type.
- Children*: to define use cases of precondition, condition, post-condition, and property references.
- Reference*: to define a relationship among objects or entities like an association, dependency, realization and generalization.
- Concept link*: to define aspects of sources to the base concepts.

Language workbenches are the tools used for meta model construction without a need to use an external parser for enabling error free context such as using JetBrains Meta Programming System (MPS) (see Figure 8), and with a need to use an external parser to debug the error messages such as using an Xtext BDD DSL language (see Figure 7). This Xtext

model has three main sections to edit and parse the DSL consistency and completeness to define a meta-model first (Abstract Syntax (CS) first) and match with a top-down approach. Such DSL style is also a bottom-up approach as it supports concrete syntax first. These sections are: Model, Parameters, and Constraints. Model is a *CloudMobilityService*. Parameters define abstract data types such as Numbers: *SLength* and *eLength*, Enumerative: *mode*, and Boolean: *routeId*, *route*, *arrivaltime* and *arrive*. Constraints are the rule-sets to check Parameters scope with values in the range of data types. So, Xtext modeling uses a textual plain language and does not support the GUI language oriented style. Likewise, the MPS follows an Abstract Syntax (AS) meta-model approach in language oriented programming style. This language style enables to create software with a set of DSLs, while language workbenches are a set of tools that apply these styles using MPS. This JetBrains MPS plugin is a typical example that allows DSL language using a projection editor for creating both persistent GUI and textual representation. The differences in between the two workbenches (e.g., MPS and Xtext) are following points (see Table 2):

Table 2. Capability Difference Between Projection and Parsing.

Capability	Concrete Syntax	Abstract Syntax
Parsing	Xtext	Xtext
Projection		MPS

```

.....
@JsonInclude(JsonInclude.Include.NON_NULL)
@JsonPropertyOrder({
    "id",
    "email",
    "lastLogin",
    "registrationDate",
    "role",
    "clusters",
    "birthday",
    "cellphoneNumber",
    .....
})
public class Properties implements Serializable,
Parcelable{
    .....
    protected Properties(Parcel in) {
        this.id = ((Id)
in.readValue((Id.class.getClassLoader())));
        this.email = ((Email)
in.readValue((Email.class.getClassLoader())));
        this.lastLogin = ((LastLogin)
in.readValue((LastLogin.class.getClassLoader())));
        ;
        this.registrationDate = ((RegistrationDate)
in.readValue((RegistrationDate.class.getClassLoa
der())));
    }
    .....
}

this.role = ((Role)
in.readValue((Role.class.getClassLoader())));
this.clusters = ((Clusters)
in.readValue((Clusters.class.getClassLoader())));
this.birthday = ((Birthday)
in.readValue((Birthday.class.getClassLoader())));
this.cellphoneNumber =
((CellphoneNumber)
in.readValue((CellphoneNumber.class.getClassLo
ader())));
    .....
}
.....
@Override
public void writeToParcel(Parcel dest, int
flags) {
    dest.writeValue(id);
    dest.writeValue(email);
    dest.writeValue(lastLogin);
    dest.writeValue(registrationDate);
    dest.writeValue(role);
    dest.writeValue(clusters);
    dest.writeValue(birthday);
    dest.writeValue(cellphoneNumber);
    .....
}
.....
}

```

Figure 4. Example: Jackson ObjectMapper Reading JSON Properties Object from JSON Model to Write Java Mode (Partial View: a JSON conversion which is generated from a NEMO customer read API JAX-RS Get HTTP operation, i.e., [https://schaufenster.informatik.unioldenburg.de:\[port\]/BlackGhostAPI/customer/read](https://schaufenster.informatik.unioldenburg.de:[port]/BlackGhostAPI/customer/read) in [34], to POJO Jackson Rest Service)



```

1  Model CloudMobilityService
2
3  Parameters:
4      Numbers sLength { 0 36 };
5      Numbers eLength { 0 36 };
6      Boolean arrive;
7      Enumerative mode { BIKE BUS WALK };
8      Boolean routeId;
9      Boolean route;
10     Boolean arrivaltime;
11     Numbers rLength { 0 36 };
12     Numbers tLength { 0 25 };
13     Boolean lDigits;
14     Boolean lSpecial;
15     Boolean lLetter;
16 end
17 Constraints:
18     # arrive==true and sLength==36 and eLength==36 and tLength==25 and rLength==36 => routeId==true #
19     # arrive==false and sLength==0 and eLength==0 and tLength==0 and rLength==0 => routeId==false #
20     # tLength==0 => (lDigits==false and lSpecial==false and lLetter==false) #
21     # rLength==0 => (lDigits==false and lSpecial==false and lLetter==false) #
22     # eLength==0 => (lDigits==false and lSpecial==false and lLetter==false) #
23     # sLength==0 => (lDigits==false and lSpecial==false and lLetter==false) #
24     # routeId==false => route==false #
25     # routeId==true => route==true #
26     # route==false => arrivaltime==false #
27     # route==true => arrivaltime==true #
28 end

```

Figure 5. Example: Descriptions of Xtext Input Domain Based on Feature Model for NEMo Customer Account Search Space excerpted from [42].

MPS tool embraces the Java to create a Java source code from the specification model. This tool enables a simple transformation based on a java source code generation. The typical transformation is important to map the rules which define the concepts to be processed and reduce the rules which remove source code and replace it with associated template. In Figure 6, the *MPSProject* contains three main key words to define the structure of template. These keywords are: *abstract class*, *extends*, and *interface*. *abstract class*: defines the concept *UseCase*. *extends*: defines the *BaseConcept* language to extend the concept *UseCase*. The *extends* keyword uses a notation symbol `<- ->` association. A typical example of concept *UseCase* refers to the customer class. The customer class as *BaseConcept* language composes the concrete *Login* class. This concrete class contains two mandatory fields: *username* and *password* with *String* abstract type. Each field has getters and setters methods based on principles of standard OOP design pattern implementation [36]. The getters are shared operations for the *Login* class which offers a response of its current object. This class has the interfaces to be implemented for this purpose. *NamedConcept*: implements the *interface BaseConcept* language for the *UseCase* customer class abstraction by writing the assertions on operations for the *Login* class. This *interface* builds on MPS template to associate with template classes. This template layout embeds built-in classes' *cellLayout*, *properties*, *children*, *reference* and custom class *customer* relationships which allows writing the java source code for *NEMoMobilityService* application.

These classes as classifier nodes that extend other necessary concrete classes. Using JetBrains MPS template each concrete class has an implementation phase. For

example, *properties* class declares the fields "id", and "routeid" with *String* and "datetime" with *date* abstract data types respectively. *MPS template extension*: this extension has capability to use external concrete java classes such as *Login*, *Location*, and *Mode*. The external class *Location* is part of class *cellLayout*, *properties*, *children*, and *reference*. This external class has fields that require MPS template model transformation with *Boolean* cell layout value in the *cellLayout* class, *double* with fields "latitude" and "longitude" in the *properties* class.

The *NEMoMobilityService* implements a field "baseurlpath" as a *String* in the *properties* class to associate with the API end point. In the case of *Mode* as external class has enum class with list of values: *BUS*, *BIKE*, *CAR*, *TRAIN*, and *TRAM* in the *properties* class. Object concept links: this template part helps to build a reference class concept lists to relate the external concrete class.

The *reference*, *cellLayout*, and *properties* associate with a class *children*. *children* class extension: includes the test cases to embrace with *usecasePropertyReference*, *usecasePrecondition*, *usecasePostcondition*, and *usecaseScenario*. This extension part such as *usecaseScenario* helps to extend a Gherkin BDD test step meta model transformation. Reducing and mapping rules: this MPS meta tool part has capability to transform an external coexisting domain models with the rules of removing and mapping constraints. A typical example is the *usecase Scenario* class with *remove* and *add methods*. Thus, the meta model idea is vital to redefine the complex system into simplified one with behavior-oriented language programming. By enabling the behaviors a pattern action class implements Gherkin BDD syntax (Given-When-Then).


```

1  "@startuml name"
2  package MPSproject <<Folder>>{
3      abstract class UseCase
4      UseCase --> BaseConcept
5      BaseConcept -- conceptLink: refers >
6      UseCase -- customer
7      customer *-- Login
8      interface INamedConcept
9      BaseConcept -() INamedConcept
10     INamedConcept "0..*" -- "1..*" cellLayout: uses >
11     INamedConcept "0..*" -- "1..*" properties: uses >
12     INamedConcept "0..*" -- "1..*" children: uses >
13     INamedConcept "0..*" -- "1..*" reference: uses >
14     INamedConcept "0..*" -- "1..*" customer: defines >
15     properties o-- Location
16     properties o-- Mode
17     (INamedConcept, cellLayout) .NEMoMobilityService
18     (INamedConcept, properties) .NEMoMobilityService
19     (INamedConcept, children) .NEMoMobilityService
20     (INamedConcept, reference) .NEMoMobilityService
21     (INamedConcept, customer) .NEMoMobilityService
22     class NEMoMobilityService{
23         String baseurlpath
24     }
25     class properties{
26         String id
27         String routeid
28         date datetime
29     }
30     class cellLayout{
31         boolean cellval
32     }
33     class reference{
34         object objLists
35     }
36     class Location{
37         double latitude
38         double longitude
39     }
40     class customer{
41         String username
42         String password
43     }
44     class Login{
45         getUsername()
46         getPassword()
47         doCredential()
48     }
49     enum Mode{
50         BUS, BIKE, CAR, TRAM, TRAIN
51     }
52     conceptLink <|-- reference: relates >
53     customer --* NEMoMobilityService: uses >
54     children *-- usecasePropertyReference: 0..*
55     children *-- usecasePrecondition: 0..1
56     children *-- usecasePostcondition: 0..1
57     children *-- usecaseScenario: 1..*
58     NEMoMobilityService -- usecaseScenario: adapts 0..* <
59 }
60 class usecaseScenario{
61     remove()
62     add()
63 }
64 @enduml
65

```

Figure 6. MPS Project Layout Implementation Textual Template to Conceptualize Idea in MDE which is adapted from [1, 6, 10, 42].

3.3. Behavior-Driven Development

In agile software team development, a behavior-driven development (BDD) is a test-first principle where test steps are scripted for running the scenarios on the system under test. By using BDD the executable behaviors or test cases for each test step is derived from the user stories and stated those

using ubiquitous language through employing a pattern matching approach. This derivation allows the stakeholders involving in the definition of user stories, which are a basic design input requirements. It is unlikely to other Test-Driven Development (TDD), BDD confirms to the user acceptance tests based on stories or requirements rather than unit tests.

The TDD performs testing on low level by refactoring the

source code of the individual program. Whereas, the BDD ensures the users acceptance which check the capability of system whether their needs are fulfilled or not with respect to a business value is tested. Naturally, this value embeds an executable pattern with Given ((preconditions: input (s)), When (condition (s) to be met) and Then (outcomes) actions. Once the Gherkin is prepared from the user stories, a BDD frameworks support several programming languages, for example, Cucumber (Java) or RSpec [9] (Ruby). As a crosswalk common element, those frameworks enable the DSL for defining scenarios, and a way to derive executable test code by parsing and evaluating pattern matching, usually the regular java expressions. One way of such DSL which can be applied in several frameworks is called Gherkin Languages [9]. Figure 7 represents a sample exemplary usage of Gherkin. The feature file consists of scenarios of the following three elements of ubiquitous language: (a) the keyword GIVEN_AND sets up the precondition(s) test environment by creating the specific context which is useful to run the test. (b) The keyword WHEN_AND validates the parameters inputs and executes the functionality under test.

(c) Lastly, using keyword THEN, the output of previous step is compared against the expected one.

Feature: Route Details Plan

In order to avoid errors the user must be able to find route details for the mobility service

Scenario: Find Route Details With Credentials

Given login API opens dialog box
And login prompts to enter credentials
When user enters user name
And user enters password
And user enters base URL path address
And press button Send in order to transfer state
And user valid login credential successful
Then system displays message successfully

Scenario: Logout Route Details

When system terminates the application
Then message displayed logout Successfully

Figure 7. Example: Two Scenarios from the Route Details NEMo Project, expressed in Gherkin Syntax (Given-When-Then).

```
...
import org.openqa.selenium.firefox.FirefoxDriver;
import cucumber.api.java.en.Given;
import cucumber.api.java.en.Then;
import cucumber.api.java.en.When;
public class Test_Route_Steps {
    public static WebDriver driver;
    @Given("^User route path is on Home Page$")
    public void user_route_path_is_on_Home_Page() throws Throwable {
        String baseUrlPath="http://schaufenster.informatik.uni-oldenburg.de:8181/BlackGhostAPI
        /senseifindroute/details?route=33b9f7f6-efe5-429f-8bc8-32fb567b9eed";
        System.setProperty("webdriver.gecko.driver", "//home/rahel/Videos/jar files/alogging/geckodriver");
        driver=new FirefoxDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.get(baseUrlPath);
    }
    @When("^User prompts to login Page$")
    public void user_prompts_to_login_Page() throws Throwable {
        driver.findElement(By.xpath(".*[@id='account']/a")).click();}
    @When("^User enters UserName and password$")
    public void user_enters_UserName_Password() throws Throwable {
        driver.findElement(By.id("log")).sendKeys(username);
        driver.findElement(By.id("pwd")).sendKeys(password);
        driver.findElement(By.id("login")).click();}
    @Then("^Message displayed Login Credentials Successfully$")
    public void message_displayed_Login_Credentials_Successfully() throws Throwable {
        System.out.println("Login Successfully"); }
    @Then("^Systems displayed route details in JSON $")
    public void Systems_displayed_route_details_in_JSON() throws Throwable {
        System.out.println("JSON data displayed Successfully");}
    @When("^User Logout from the Application$")
    public void user_Logout_from_the_Application() throws Throwable {
        driver.findElement(By.xpath(".*[@id='account_logout']/a")).click();}
    @Then("^Message displayed Logout Successfully$")
    public void message_displayed_LogOut_Successfully() throws Throwable {
        System.out.println("Logout Successfully");}}
```

Figure 8. Example: Code Equivalence for the scenarios in Figure 7, 9 and 15, expressed in Java Cucumber.

Each step in turn, is backed by a step definition code template, that converts the text into executable program code using regular expressions, e.g., to extract method parameters. Figure 7 shows the test step templates needed to execute both scenarios shown in Figures 8 and 9. As each step is individually backed by support code, steps can arbitrarily be

combined to create new acceptance tests. Gherkin syntax is capable to comprehend the use of natural language; scenarios can also easily be created by non-programmers, e.g. the future users of the software themselves. In that manner, a ubiquitous language is created, that improves communication between all stakeholders involved in the software

development process, for example, owner, developer, test engineer, and customer.

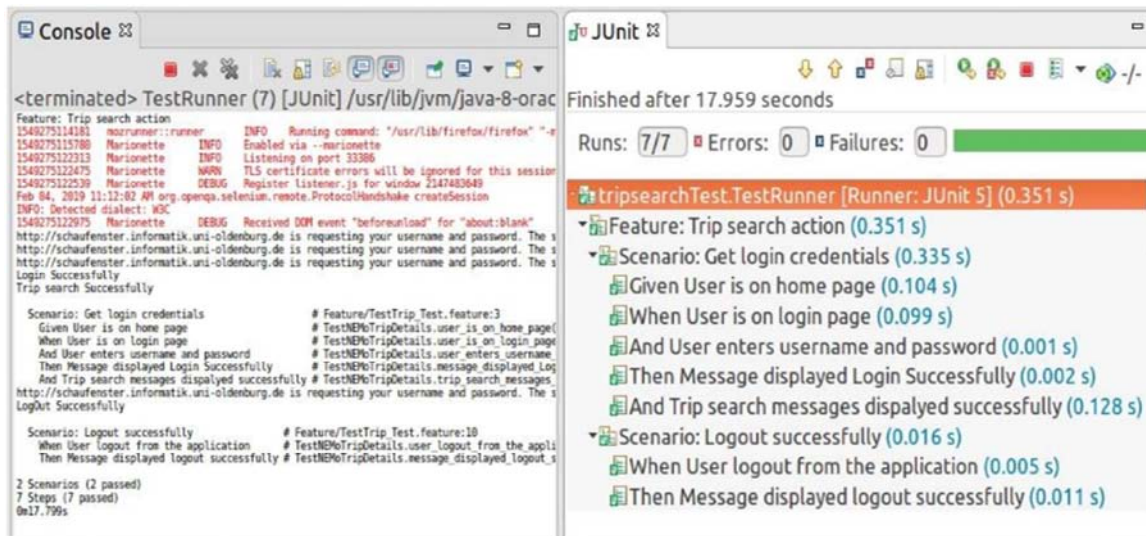


Figure 9. DSL Gherkin Model Cucumber Junit Java Tests in Eclipse Modeling Framework.

4. Tools Related to Approach Formation

The tools related to the approach formation include supporting open source automation frameworks, model consistency checking, modeling UML and REST service model.

Open source automation frameworks: open library frameworks are popular for supporting many research works to be reusable as they have been evidenced by its previous scientific findings on various applications. These libraries integrate easily within the framework for designing, building and validating the software products with model-driven software engineering principles. Notable examples are *QF-Test*, *Jenkins*, and *cucumber* automation frameworks.

(a) *QF-Test* [31]: using QF-Test testers have capability to use Graphical User Interface (GUI) for functional realization in system testing of Java Swing, JavaFX, Eclipse and Remote Call Procedure (RCP) applications and cross-browser web testing. Its stability, flexibility and accessibility enable an easy integration with other continuous delivery automation framework like a Jenkins. Above all, QF-Test becomes popular in the quality assurance due to its striking characteristics that make it dominant in the testing team from the crowd. In Figure 10, *BDDQFTestSystem* defines *QFTestRunner* which composes two main tasks: *TestSuites* and *QFTestJenkinPlugin*. *TestSuites* contains following core functionalities in this regard. The goal of QF-Test in terms of following points: (i) *Test execution*: this capability grants to get test preparation for easy application integration with only necessary components, and manage the dependency test steps which enable independent executable test cases including automatic error handling. (ii) *Flexible data drivers*: as test data is to reuse multiple times, the data drivers allow using a flexible integration of internal or external data driven testing, for example, from comma separated value (CSV), or excel file. (iii) *Usability*: by using this capability, test engineers

enable to use simply their previous knowledge with low level programming skills for standard usage. (iv) *Test suite layer formation*: creation of such a layer makes clear tree view and user friendliness which enable an easy test case handling with its testing principle by capture and replay for quick debug and log-run generation. (v) *Feature*: QF-Test supports several features. Out of these, it includes platforms, web drivers, multiple technologies in one system, and handles keyword for behavior driven testing through planning a document on the REST specification. (vi) *BDD framework support*: this part mainly embraces external components of BDD formation. Examples: *TestSuites* defines BDD concept such as *TestSet* to include scenario which maps with *TestCase* to compose *TestStep* from Feature. This *TestStep* initiates *WebDriver* which calls upon *BaseURLPath* to execute the application under test, that is, SUT.

(b) Jenkins QF-Test plugin continuous automation: the QFTest enables plugin tool such as Jenkins continuous delivery framework to enhance quality of software coverage analysis for ensuring end to end continuous integration testing. QFTestRunner framework supports plugin configuration for Jenkins build tool analysis on the specified SUT.

(c) Graph Markup Language (GraphML): this language has capable to extend data exchange format such as XML and JSON which offer platform and language independent service oriented architecture in a cloud based system. GraphML has a versatile functionality which helps to adapt the complex system into simplified one using its capability composing a graph service such as a standard web services. Web service is a component that embraces a complex design system such as cloud service to provide abstraction that bridges the AppServer and the application. This service has a graph oriented architecture where each node in the graph connects to its edge with a data label [13, 38].

Thus, the design of GraphML aims to satisfy following goals in mind: (i) Easy parsing and interpreting format for

both human and machine. That is, no unclear instructions which are a well-formed interpretation for each valid GraphML path generation. (ii) Expressible within the same basic format, which avoids the limitations with respect to a graph model, e.g., hypergraphs, hierarchical graphs. It ensures the generality. (iii) Extensibility: It allows extending

the format in a well-defined way to present other data needed by any arbitrary applications or complex use. (iv) Robustness: It enables the systems capable of handling the complete range of graph models or mapping information to perceive and extract the subset it handles.

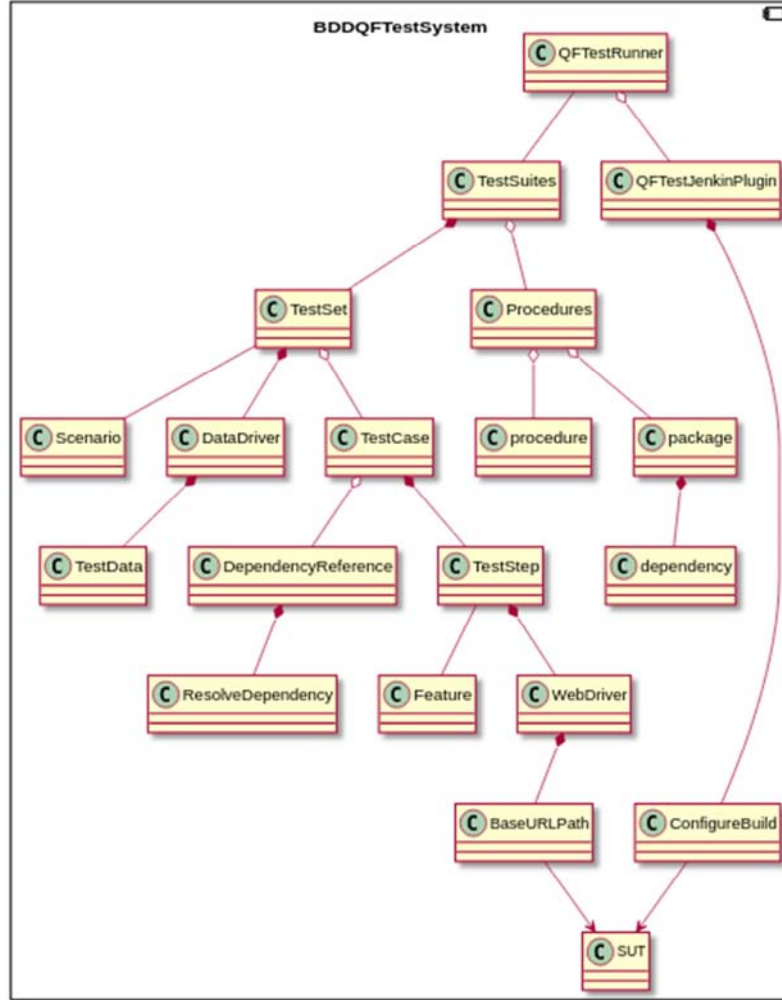


Figure 10. QF-Test BDD Design to Test Specification Model: Adapted from [32] using [30] UML Technique.

Graph model: the graph model used in this paper is labeled in a digraph, i.e., a tuple, $G = (V, E, D)$, where V is a set of nodes, E a set containing directed edges, and D a set of data labels that are partial functions from $\{G\} \cup V \cup E$ into some specified range of values. The data labels encodes, e.g., properties of nodes and edges such as graphical variables. To illustrate a bit about the concept of this graph, referring Figure 13 is important. Since GraphML supports extension from one format to another which makes it expressible and simplify in service accessibility and maintainability through model transformation language. It is also possible to re-engineer in the reverse way from JSON to GraphML format. Accordingly, we generate the GraphML from Figure 13. The syntax used in this case is:

```
$ java -jar graphwalker-cli-4.3.0-SNAPSHOT.jar convert -i /$USER_HOME/.../NEMoMobilityGraph-Model.json -f graphml.
```

5. Approach Formations

This part forms the approach through functional realizations. This realization understands the use case based on the stakeholders and requirements defined on section 2.3. Following a detailed UML gherkin meta model is enabled through AS first approach based on concepts in MPS template discussed on section 3.2.

5.1. Realization of Enumerated Functionality

This section aims to understand functionality based on various behavior-driven software engineering realization facilities. The main ones that are covered in this part are as follows: User modeling, Gherkin Meta model Realization, and Finite State Machine. *User modeling* defines the use case

diagram based on the expected requirements in section 2.3 which is indicated in Figure 11. This modeling consists of (a) (*Owner*) contexts the stereotype <<UbiquitousLanguage>> to the stakeholders' requirements through portable natural

language which is very close to the user stories. (b) (*BusinessAnalyst*) describes the stereotype in the idea around <<UserStoryDefinition>> to capture the natural language.

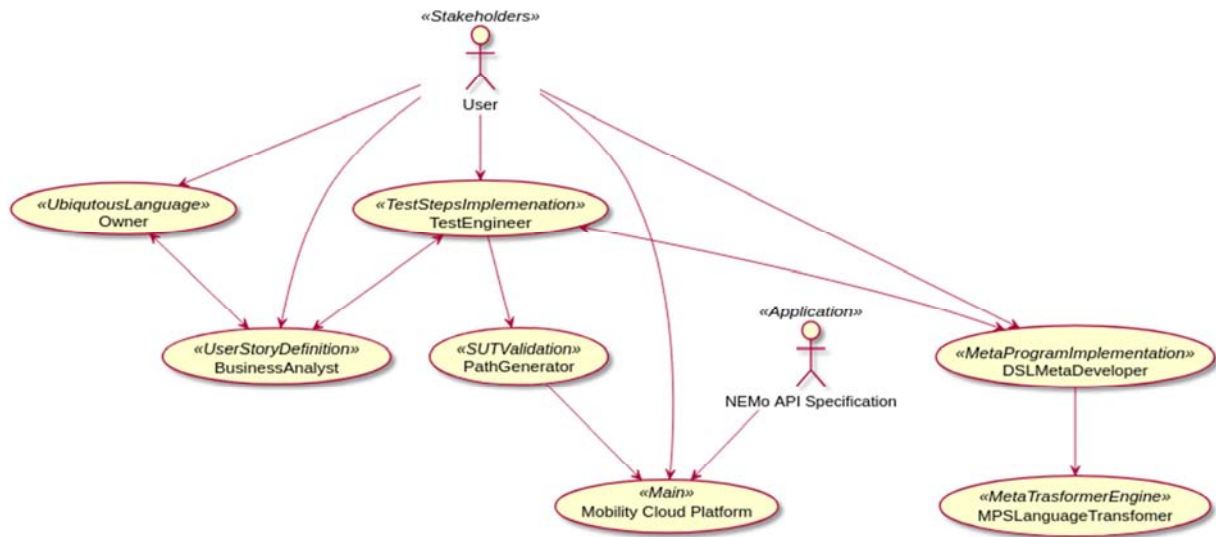


Figure 11. UML Use Case Modeling for Stakeholders Realization.

(c) (*DSLMetaDeveloper*) describes the stereotype in the idea around <<MetaProgramImplementation>> to design small specific language. (d) (*MPSLanguageTransfomer*) generates the source code to define DSL meta template in the idea around <<MetaTrasformerEngine>> and transform and correct for enabling DSL instances to be executable. (e) (*TestEngineer*) writes cucumber tests based on generated DSL meta template in the idea around <<GherkinBDDTestScripter>> to enable test coverage analysis. (f) (*TestEngineer*) writes cucumber tests for generated DSL meta template in the idea around <<GherkinBDDTestScripter>> to enable test coverage analysis

based on the stereotype implementation for <<TestStepsImplementation>>. (g) (*PathGenerator*) is an algorithm engine that generates paths through a model with the idea of keywords stereotype around <<Keyword-DrivenGeneration>>. This (*PathGenerator*) helps to describe in the stereotype idea around <<SUTValidation>> to validate the (*Mobility Cloud Platform*) as SUT to abstract the application which is a NEMo API Specification. *Gherkin Meta model with GraphWalker MBT* illustrates the examples to define meta functional realization based on user stories (see Figure 12).

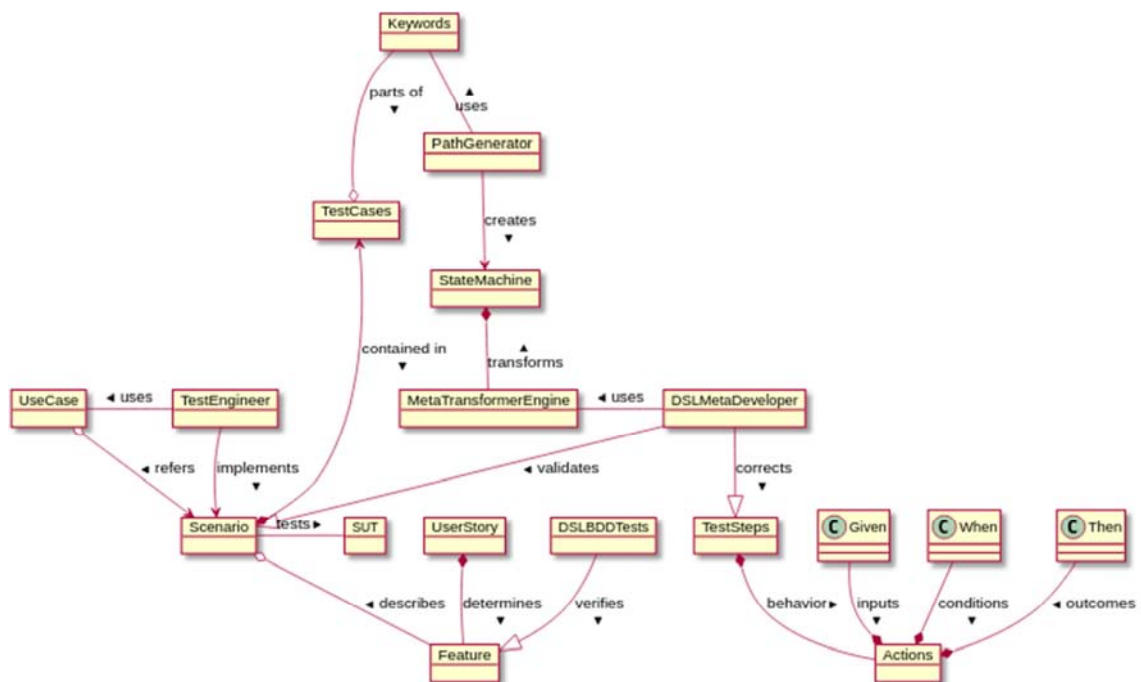


Figure 12. UML Gherkin BDD Meta Model Diagram for Functional Realization.

GraphWalker Test Path Realization Process: this test realization for the FSM can be ensured in two ways in this approach formation. One is to design a digraph model using

[illegible]

Generator is an algorithm that enables to decide how to traverse a model. Generators vary to create different test sequences and navigate in various ways. Examples of generator algorithms: *Random*, and *A Star*.

- vertex or edge using the A* search algorithm [44]. Syntax: *a_star* (<stop_conditions>). This algorithm must use a stop condition that names a vertex or an edge. Examples: walks the shortest path to the vertex *v_Logout* and then stops. Example: *a_star* (reached vertex (*v_Logout*)) - generate a shortest path through the model to the vertex *v_Start* (see Figure 17). Here, only the elements of the shortest path from *v_Start* to *v_Logout* have been colored green, meaning they have been traversed. A Star combines the heuristic function to evaluate the cost of each path and determine the admissible best route to reach the goal based on the test

6. Application: NEMo Mobility Service

Exemplary behavior-driven mobility service for planning document on application of NEMo project is adapted for model consistency checking through meta-model approach. Once this adaptation ends a DSL source code is ope-rationalize for model consistency checking. Then, we build the test steps that bind an application through service end point, which is applicable for each feature to test the mobility, has to be exercised for complete coverage on the SUT. However, in practice it is infeasible to cover all the nodes for all service end points rather a sample instance of an end point has to be examined in the experimental analysis [11]. In [1, 42], the authors have illustrated high level descriptions with examples to define the NEMo mobility core services. This work comes with a new option that specifies a solution using graph based meta modeling concepts and tools to examine and explain the quality of software coverage analysis. For this purpose, the technical NEMo Route plan system is referred [34]. The REST API parameters required to call mobility services are the following points: (a) *String routeid* field as indicated in MPS template for properties class under MPSProject. This field takes the id value of Google Map which is a Geo-reference of the object

Location by evaluating the coordinate points with String data type for latitude and longitude attributes. The minimalist Google Map route system in the vertical prototype applies how to use *tripRequest* transportation mode to suggest the routes. This idea helps to set the API call for requesting the route details information. (b) *String baseUrlPath*: the API application repository host machine helps to represent the path for route details from the AppServer with support of Google web service (see Figure 8). In the same manner, in Figure 16, the route provider from this web service helps to offer the *routeids* for selecting the best route given the variable *transportModes* to take such as "car" and "bus" values based on the *tripRequest* in terms of parameters: *time*, *start location* (or origin), and *end location* (or destination). After the route data is collected with *FindRoute* service via Google map, the concatenated *travelInfo* for each routeid is calculated to determine the best routes through coalesce service as RouteSelector out of concatenated route id for enabling *suggestedRoutes*. (c) *String baseUrlPath*: the API application repository host machine helps to represent the path for route details from the AppServer with support of Google web service (see Figure 8).

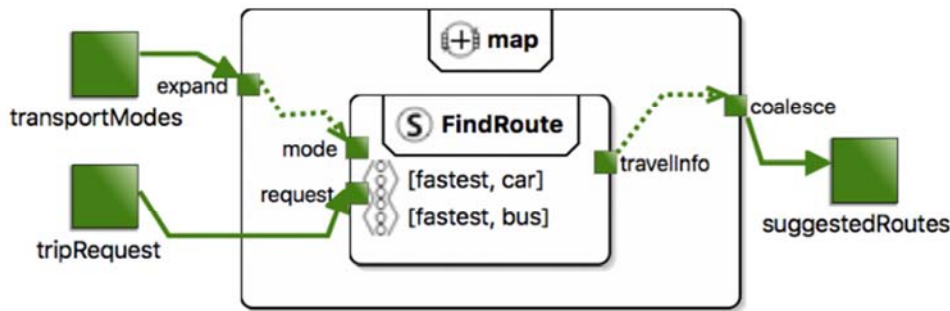


Figure 16. Minimalist Google Map for Mobility Route System excerpted from [24].

```

//The sequence in a unit test which is written in java.
@Test
public void NemoRoutePlan_Smoke_Test(){
    v_Start();
    e_StartAPIDialog();
    v_RestBaseUrlPath();
    e_SendRestAPICall();
    v_RestAPICall();
    e_enterAccount();
    v_LoginPrompted();
    e_ValidLoginAuthenticated();
    v_ValidLoginCredential();
    e_VerifyResource();
    v_VerifiedValidState();
    e_ViewResourceSuccess();
    v_ViewResourceInJson();
    e_SystemLogout();
    v_Logout();
}
  
```

Figure 17. Current Element Test Path Complete Coverage Functional Realization.

7. Evaluation

The evaluation phase covers a detailed analysis of keyword test sequence from the model in Figure 13. A prototype model is made to write a java source code based on

the keyword generated (see Figure 18, and Figures 25-29). Furthermore, use of QF-Test helps to understand the dependency reference validity call, report generation from run-log trace, and visualize test coverage analysis. Finally, the report comparisons and the test case automation on test requirements are implemented.

7.1. Keyword Generation to FSM Mode

Test path generation usually results in keyword formation based on the Gherkin BDD syntax to apply the BDD styles. In Figure 17, the current element names are generated optimally with an algorithm *a_star* generator using condition *v_Logout* for stop state operation. This generation helps to write tests for each state of keywords to verify the SUT. In Figure 18, implementing each keyword enumerated operation is a useful way to verify the system through an end to end integration testing, which maximizes the optimal coverage percentage, and a unit test is also possible through assertion for each state to attain a 100% coverage analysis on the specified service. This way, considering a retrieval of all end points for the specification model via a REST client service like a Postman tool is important to define the entire test coverage, but it is a

duplicated effort and costly [11]. However, taking a sample instance of model path is helpful to generalize all with an optimal generation algorithm such as *a_star* generator using condition *v_logout* for stop state operation.

```
//The sequence in a unit test which is written in java.
@Test
public void NemoRoutePlan_Smoke_Test(){
    v_Start();
    e_StartAPIDialog();
    v_RestBaseURLPath();
    e_SendRestAPICall();
    v_RestAPICall();
    e_enterAccount();
    v_LoginPrompted();
    e_ValidLoginAuthenticated();
    v_ValidLoginCredential();
    e_VerifyResource();
    v_VerifiedValidState();
    e_ViewResourceSuccess();
    v_ViewResourceInJson();
    e_SystemLogout();
    v_Logout();
}
```

Figure 18. Test Sequence to Java Units with States and Edges for end to end Smoke NEMo Route Details Test Coverage.

7.2. QF-Test and Jenkins Build Analysis

This section covers the idea around QF-Test implementation and reporting to compare with QF-Test Jenkins build configuration and visualization. Each figure under mentioned is taken from HTML report automated generation which is after the test sequence dependency

reference validity calls (see Figure 19, and Figure 20).

1) *QF-Test Implementation and Visualization*: the idea is to represent the implementation and visualization based on the summary of report for QF-Test node test suites, *NEMoRouteDetails.qft*. *QF-Test Test Suite Validation Calls*: this QF-Test validation call for this test suite is important to analyze three primary functionalities: *show valid*, *show invalid*, and *show not checked*. In Figure 19, all test sequences which are available in the hierarchy for test set and test cases including data driver, test data, procedures with test steps and package are validated. This validation process shows 21 *valid Ok* status, zero no invalid, and zero not checked calls. *QF-Test Report Generation*: after the validity is completed for all QF-Test design run-log, the report generation is made from it. In Figure 20, using run-log extra menu a report generation dialog box is displayed. This box has a list of check boxes that refer to the active tasks for processing this report. The typical ones are to create HTML report, create Junit report, list Test Steps, list exceptions, list warnings, create xml report, list errors and so on. Accordingly, HTML report is generated under certain directory with 20 scale factor for thumbnails in %. The specified report directory path is:

\$ /root/.qftest/QF-Testid/QF-Testid_report.html, So, the generated QF-Test ID is 191230183116. Then, the report is called by the browser using a Firefox Mozilla, and the report representation is depicted below with the topic coverage report and visualization.

Idx	Reference	Referring node	Container node	Suite	Result
9.1	startStop.startSUT	Call procedure: startStop...	Dependency: sutStarted	NEMoRouteDetailsBDI.qft	Ok
10.1	startStop.stopSUT	Call procedure: startStop...	Dependency: sutStarted	NEMoRouteDetailsBDT.qft	Ok
11.1	qfs.utils.variables.deleteAllGl...	Call procedure: qfs.utils.va...	Dependency: sutStarted	NEMoRouteDetailsBDT.qft	Ok
12.1	qfs.utils.variables.deleteAllGl...	Call procedure: qfs.utils.va...	Procedure: stopSUT	NEMoRouteDetailsBDT.qft	Ok
13.1	NEMoRouteDialog	Wait for component	Procedure: login API open...	NEMoRouteDetailsBDT.qft	Ok
14.1	username	Mouse click: (122,19)	Procedure: user name is s...	NEMoRouteDetailsBDT.qft	Ok
15.1	username	Input: *\$(username)*	Procedure: user name is s...	NEMoRouteDetailsBDT.qft	Ok
16.1	password	Mouse click: (40,19)	Procedure: user pass wor...	NEMoRouteDetailsBDT.qft	Ok
17.1	password	Input: *\$(password)*	Procedure: user pass wor...	NEMoRouteDetailsBDT.qft	Ok
18.1	OkButton	Mouse click	Procedure: press button ...	NEMoRouteDetailsBDT.qft	Ok
19.1	RouteDetailsTable	Check text: item \$(usem...	Procedure: value in colum...	NEMoRouteDetailsBDT.qft	Ok
20.1	RouteDetailsTable	Check text: item \$(passw...	Procedure: value in colum...	NEMoRouteDetailsBDT.qft	Ok
21.1	RouteDetailsTable	Check items: row (2)	Procedure: row fills with <...	NEMoRouteDetailsBDT.qft	Ok

Results: 21 of 21 match(es) found

Set mark for selected nodes: Blue, Red, Yellow, Green

Details: Valid calls: 21, Invalid calls: -, Not checked calls: -

Figure 19. Test Sequence Dependency Reverence Validity Calls for Node Test Suites: *NEMoRouteDetails.qft*.

QF-Test Test Coverage Report and Visualization: the test coverage and visualization tasks contain a summary report which includes a pie-chart with built-in various colors to show the different outputs. The report summary for test suites has nine agendas with different coverage functionality. With this respect, these agendas have its own color to name the coverage criteria's as indicated in Figure 21: (a) Total number of test cases which exercise the SUT. (b) Number of test cases with exception. (c) Number of test cases with errors. (d) Number of test cases with expected errors. (e) Number of successful test cases. (f) Number of expected test cases. (g) Percent test-cases passed. (h) Time spent in tests. (i) Expected run time. Each agenda computes the results in % and non-zero calculation is visible on the pie-chart with its designated icon color. This pie-chart report contains data labels under it with Failed red and Passed green icons for referring the chart. However, the chart diagram contains a shaded dimension outside the green shaded part which needs to be defined with some other way what functionality it handles. There is no failed state observed on the pie-chart. This unclear dimension is defined with QF-Test Jenkins

automation testing (see Figure 23). The aim of doing this is to examine the QF-Test framework capability through comparing with other cloud based continuous automation such as Jenkins framework.

This result also provides a feedback to the software developers for aware of the difference to enhance the product. This QF-Test HTML report has an overview of test sets: *URL SetUp* and *Scenario* with *Passed (100%)* status, and *test cases: Get route details steps* with *Passed* status in 27 seconds for functionalities to compute *time spent in tests* and in *expected runtime*. Each time spent in the tests and in the expected runtime for *URL SetUp* test set is 2 seconds. For *Scenario* test set, this time is 30 seconds performance response functionalities with *passed* status. The overall result is 2 and 33 seconds for both time spent in tests and in expected runtime on running *URL SetUp* and *Scenario* respectively. The QF-Test test steps represent each Gherkin BDD syntax for BDD tests for evaluating each response time in milliseconds. This result enables the test engineer to devise a methodology for each test step to evaluate the functional performance. This evaluation is depicted in Figure 22.

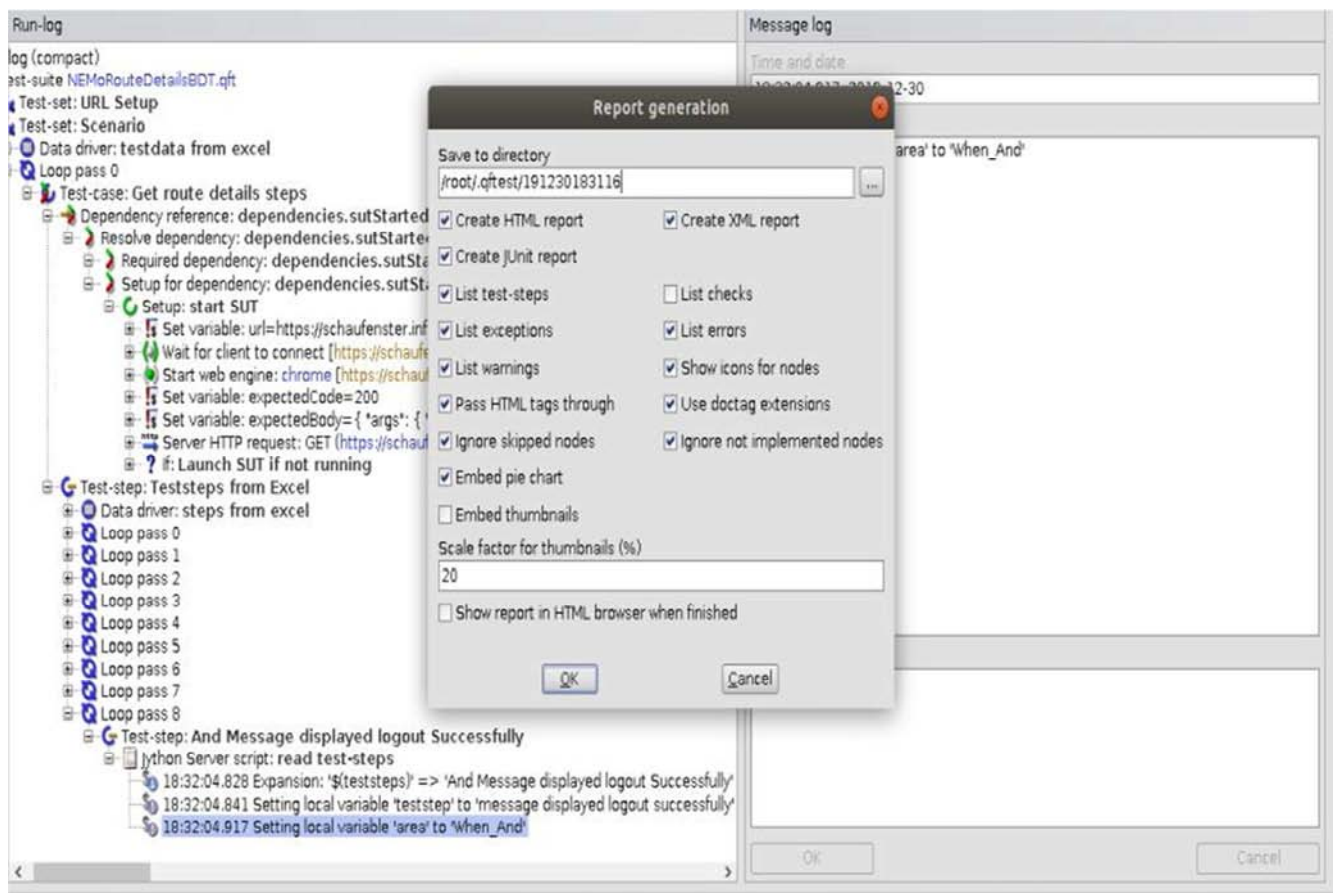


Figure 20. Test Report Generation from the Run-Log Trace Analysis for Node Test Suites: NEMoRouteDetails.qft.

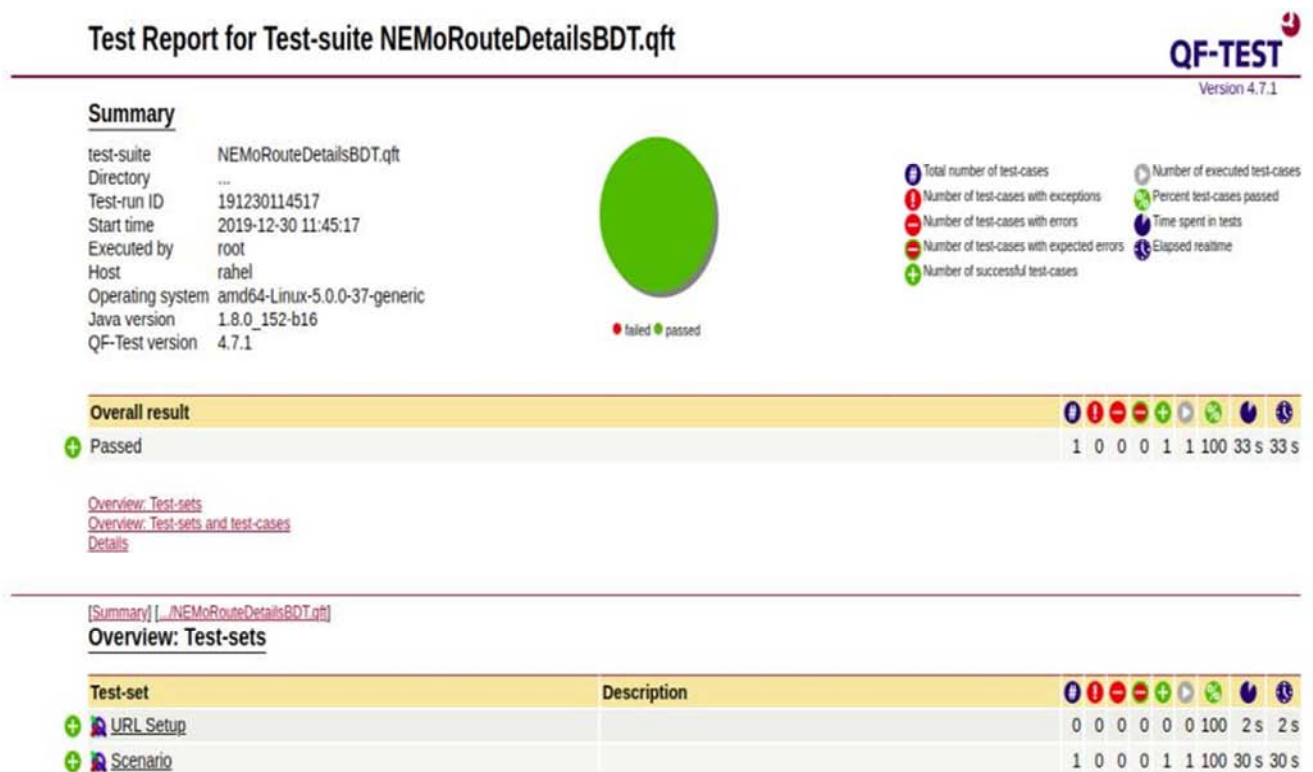


Figure 21. Summary of QF-Test Report for Node Test Suites: NEMoRouteDetails.qft.

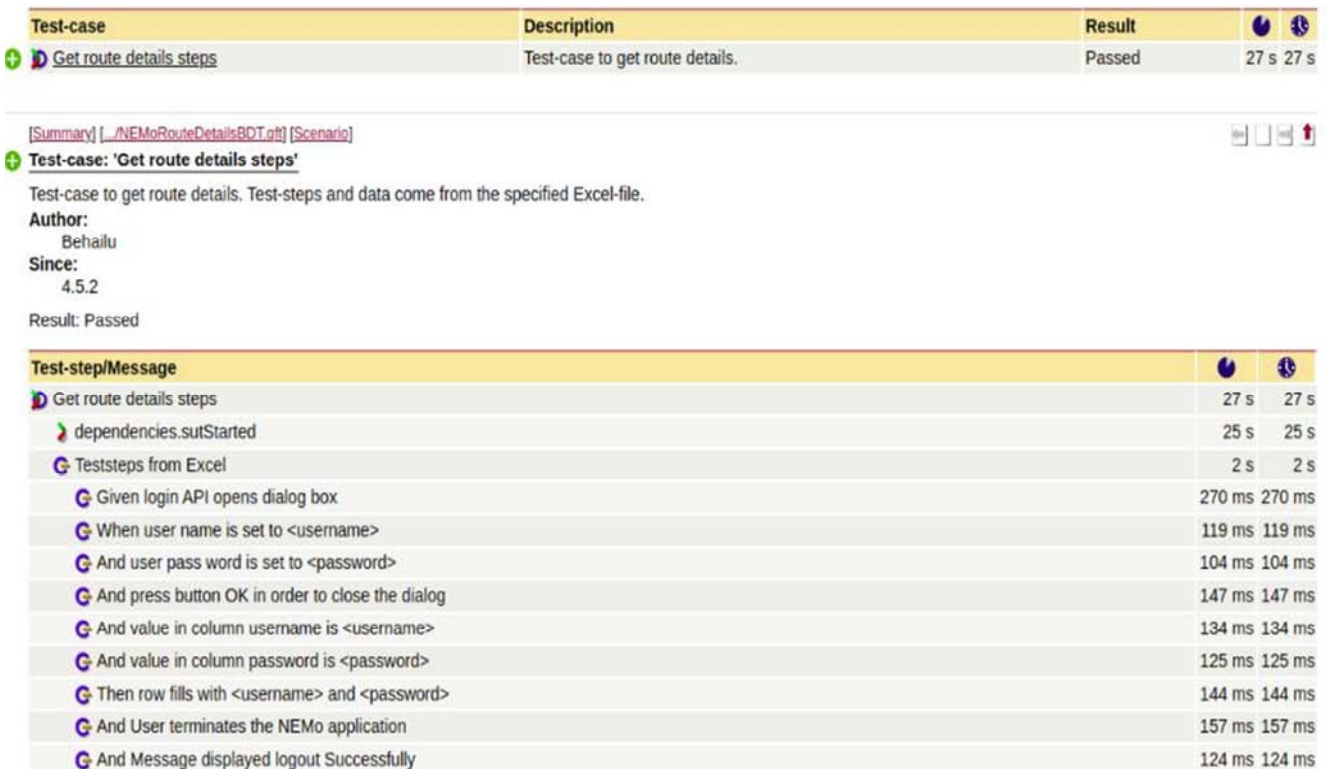


Figure 22. QF-Test Test Case Scenario "Get route details steps" and Test Steps for Node Test Suites: NEMoRouteDetails.qft.

2) *Jenkins QF-Test Plugins Continuous Build Analysis*: the continuous automation testing is important in the evaluation process to be agreed upon changes on the SUT with consistent updating. *Jenkins Build Configuration*: the Jenkins

continuous integration framework is one of the promising approach to enable a versatile third party software tools like a QF-Test and other testing libraries. The Jenkins includes these external plugin tools with a safe and resilience manner

which allow receiving external executable scripts that have already tested outside it. This capability empowers it to be one of the chosen options for continuous delivery of test automation. The Jenkins settings and configuration used in this paper are: *General*, *Build environment*, *Build action* and *Post build action*. The *Jenkins General tab* allows enabling to set use customs workspace.

This workspace refers to the SUT on local directory path: `$/opt/qftest/qftest-4.7./demo /keywords/behaviordriven/`, the Jenkins *Build Environment tab* allows configuring reliability and robustness criteria to abort the build when the time-out strategy, and timeout as a percentage of recent non-failing builds and number of builds, and timeout minutes are attained. Example: *time-out strategy= elastic* and timeout as a percentage of recent non-failing build=150% and number of builds=3 and Timeout minutes=60. If the build duration lasts longer than this percentage of three most recent non-failing builds, the build will be terminated and marked as aborted. The Jenkins *Build action tab* allows checking *QF-Test Run Test* to set as Name of test suites or Parent folder text box, for example, *NEMoRouteDetails.qft*. The Jenkins *Post Build Actions tab* allows checking *QF-Test settings: Archive the artifacts* in File to archive text box:

`/root/.qftest/*.q*` and Publish HTML Reports to set Reports with HTML directory to archive in text box:

`/root/.qftest/routedetailsreport/191230183116_report.html`, *index page: index.html*, and *Report title: HTML Report*. Using this tab, the build run actions uses the Gherkin BDD test steps to apply a *1.0* health amplification report. This number 1.0% refers to failing tests scores as 99% health which helps to decide the degree of robustness and reliability on the application under test. The result shows, after *Post Build Actions*, valid output sequences of QF-Test setup with its test steps on Jenkins console preview box as shown in Figure 24.

Jenkins Test Coverage Report and Visualization: this test coverage and visualization task contains a summary report which includes a pie-chart with built-in various colors to show the different outputs. The report summary for test suites has nine agendas with different coverage functionality. With this respect, these agendas are depicted in Figure 23 below: Each agenda computes the results in % and non-zero calculation is included in the pie-chart with its designated icon color. This pie-chart report contains data labels with Failed in red, Passed in green and not executed in gray color icons for referring the chart.



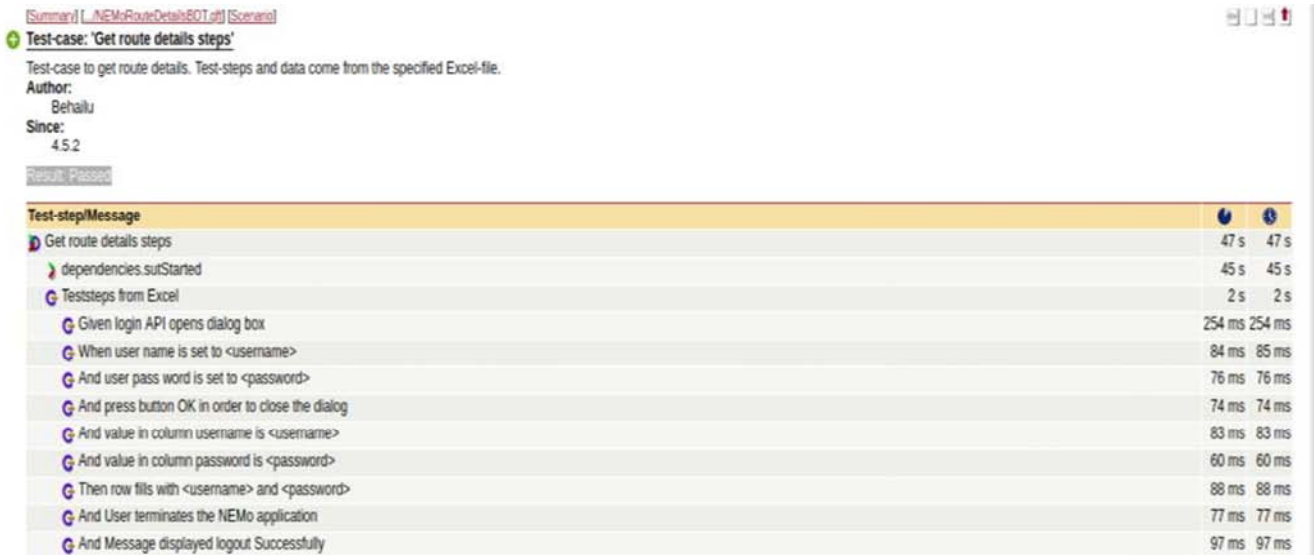
Figure 23. Summary of QF-Test Jenkins Test Coverage Report for Node Test Suites: *NEMoRouteDetails.qft*.

Unlike the QF-Test, the Jenkins QF-test chart diagram mines additional data “non-executed” test cases that was hidden in the case of QF-Test report (see Figure 22). This effort helps to compare the QF-Test framework capability with Jenkins QF-Test plugins. This result also provides a feedback to the software developers for realizing their difference and for enhancing the product. This QF-Test HTML report has an overview of test sets: *URL SetUp* and *Scenario* with Passed status and test cases: *Get route details steps* with Passed status in 47 seconds for functionalities to compute *time spent in tests and in expected runtime*. Each *time spent in the tests* and the *expected runtime* for *URL SetUp* test set is 16 milliseconds. For *Scenario* test set, this time is 54 seconds response time for both built-in

functionalities with passed status. Figure 24 depicts the required test step BDD classes based on test execution with its responses which define the time spent in tests and its responses which define the time spent in tests and expected runtime in milliseconds. Accordingly, out of nine test step Gherkin syntax, that is, *TS5*: value in column password is `<password>`, *TS5* takes 60 ms response time which is the fastest time spent in tests and expected runtime, whereas *TS0*: Given login API opens dialog box takes 254 ms which is the slowest time to finish the tests as well as the runtime expectations. Others test steps time-duration are in between these two extreme response time, that is, 60 and 254 ms. For this, the QF-Test Jenkins build run shows the run-log analysis to the SUT.

7.3. Step Step Method to Create GraphWalker Test Automation

Quality product analysis is not only limited to use a QF-Test design test automation, but also it is possible to prepare a testing strategy through step by step test case implementation. The step by step method is a test automation way which implements directly the generated keywords from DSL BDD gherkin syntax as annotated `@Test` classes using



The screenshot shows a Jenkins test case scenario titled "Get route details steps". It includes a summary, author (Behailu), and version (4.5.2). The test case is marked as "Result: Passed". Below the summary is a table of test steps with their messages and durations.

Test-step/Message	Duration
Get route details steps	47 s
dependencies.sutStarted	45 s
Teststeps from Excel	2 s
Given login API opens dialog box	254 ms
When user name is set to <username>	84 ms
And user pass word is set to <password>	76 ms
And press button OK in order to close the dialog	74 ms
And value in column username is <username>	83 ms
And value in column password is <password>	60 ms
Then row fills with <username> and <password>	88 ms
And User terminates the NEMo application	77 ms
And Message displayed logout Successfully	97 ms

Figure 24. QF-Test Jenkins Plugin Test Case Scenario "Get route details steps" and Test Steps for Node Test Suites: NEMoRouteDetails.gft.

The step by step prototype implementation used in this work is summarized as: Required External libraries, Step By Step Test Implementation and Tests for Integration, Functional and Stability. Required External libraries: usually, these libraries are located in a maven open source repository. This repository provides the easiest solution to make testing, building and deploying the software product. In Figure 25, each imported library has its own responsibilities to make verification and validation over the model given in terms of input. Example: *.EdgeCoverage helps to examine each edge on the model using condition that is given to be participated in test path coverage. *.Paths helps to get each *.Path element in the model which adds each specific path with its state and edge to the context model. In addition, each *.Path should be known via FSM library, that is, *.machine.* to context *.model.* created with *.ExecutionContext for building and running each vertex and edge from the start to the end. Step By Step Test Implementation: this step is coded using the fastest algorithm that covers test paths generated on the specified condition as stated in Figure 18. The testing framework is a testNG for enabling integration testing using `@Test` annotation class for each test step with generated keywords. Each keyword in the step by step implementation is equivalent to the Gherkin BDD syntax in QF-Test node test suites for the Scenario definition to the test cases.

testing framework. This method requires a virtual machine such as java development kit, and integrated development environment (IDE) or other command line interface (CLI) continuous tooling like an Ant and Maven. For this implementation, *TestNG* is a promising testing framework to verify each state or test paths in a model together with path dependency analysis through fully covering the smoke, stability, and functionality tests.

```
package nemo.routeapi.demo;

import org.graphwalker.core.condition.EdgeCoverage;
import org.graphwalker.core.condition.ReachedVertex;
import org.graphwalker.core.condition.VertexCoverage;
import org.graphwalker.core.condition.TimeDuration;
import org.graphwalker.core.generator.AStarPath;
import org.graphwalker.core.generator.RandomPath;
import org.graphwalker.core.machine.ExecutionContext;
import org.graphwalker.core.model.Edge;
import org.graphwalker.java.test.TestBuilder;
import org.graphwalker.java.test.TestExecutor;
import org.graphwalker.core.machine.*;
import org.graphwalker.core.model.*;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.Assert;
import org.testng.annotations.Test;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.concurrent.TimeUnit;
```

Figure 25. Typical External Libraries Used for Step By Step Test Case GraphWalker Test Automation.

In Figures 26-27, the test class implements an object *TestNemoRouteAPISearch* which extends to inherit the super class *ExecutionContext*. This context class initializes the graph mark language using a static *Paths* class through accessing a static get root directory which assigns the static *Path* to an object *MODEL PATH*. The constructor *TestNemoRouteAPISearch* implements to initialize this object model path. Moreover, each keyword is annotated

with `@Test` class as `e_RestServiceBaseURLPath`, `v_startAPIDialog`, `v_sendRestAPICall`, `e_InvalidLogin`, `e_InvalidLoginRepeat`, `e_ValidLoginRepeatAgain`, `v_LoginPrompted`, `e_ViewResourceInJson`, `e-`

`_ValidLoginAuthenticated` and `e_SystemLogout`. With this respect the step by step test execution validates all test paths effectively.

```
....
public class TestNemoRouteAPISearch extends ExecutionContext {
    public final static Path MODEL_PATH = Paths.get("/home/rahel/nemoproject/src/main/java/nemo/routeapi
        /demo/NEMOMobilityGraphModel.graphml");

    WebDriver driver = null;
    public TestNemoRouteAPISearch (Path MODEL_PATH){
        this.MODEL_PATH= MODEL_PATH; }

    @Test
    public void v_RestServiceBaseURLPath() {
        driver = new FirefoxDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.get("schaufenster.informatik.uni-oldenburg.de:8181 "+"RestServiceResourcePath()");}

    @Test
    public void e_StartAPIDialog() throws Exception{
        v_RestBaseURLPath();
        System.out.println("e_StartAPIDialog()");
    }

    @Test
    public void e_SendRestAPICall() throws Exception{
        v_RestAPICall();
        System.out.println("e_SendRestAPICall()");
    }

    @Test
    public void e_InvalidLogin() {
        driver.findElement(By.name("username")).sendKeys("incorrect");
        driver.findElement(By.name("password")).sendKeys("incorrect");
        driver.findElement(By.xpath("//input[@value='Log In']")).click();
    }
}
```

Figure 26. Step By Step Test Case GraphWalker Test Automation.

```
...
@Test
public void e_InvalidLoginRepeat() {
    driver.findElement(By.name("username")).sendKeys("incorrect");
    driver.findElement(By.name("password")).sendKeys("incorrect");
    driver.findElement(By.xpath("//input[@value='Log In']")).click();
}

@Test
public void e_ValidLoginRepeatAgain() {
    driver.findElement(By.xpath("//input[@value='Log In']")).click();
    v_LoginPrompted();
    driver.findElement(By.name("username")).sendKeys("6aabfc18-6072-4a5f-96c6-e90112c8eb4b");
    driver.findElement(By.name("password")).sendKeys("*****");
    e_ValidLoginAuthenticated();
}

@Test
public void e_ValidLoginAuthenticated() {
    v_ValidLoginCredential();
    System.out.println("e_ValidLoginAuthenticated()");
}

@Test
public void e_RestServiceBaseURLPath(){
    v_RestAPICall();
    System.out.println("e_RestServiceBaseURLPath()");
}

@Test
public void e_VerifyResource(){
    v_ValidLoginCredential();
    v_VerifiedValidState();
    System.out.println("e_VerifyResource()");
}

@Test
public void e_ViewResourceSuccess() {
    v_ViewResourceInJson();
    System.out.println("e_ViewResourceSuccess()");
}
}
```

Figure 27. Step By Step Test Case GraphWalker Test Automation.

Then, using a simple state machine the test model context is conducted for `fullCoverageTest ()` operation and each vertex is iteratively executed to verify the next step test execution with `@Test` annotation on each sample keyword operation with return

type void (see Figure 28).

```
...
@Test
public void e_SystemLogout() {
    v_Logout();
    driver.findElement(By.xpath("//a[text()='Log Out']")).click();
}

@Test
public void fullCoverageTest() {

    // Create an instance of our model
    TestNemoRouteAPISearch model=TestNemoRouteAPISearch(MODEL_PATH);

    // Build the model (make it immutable) and give it to the execution context
    this.setModel(model.build());

    // Tell GraphWalker to run the model in a random fashion,
    // until all vertexes are visited at least once.
    // This is called the stop condition.
    this.setPathGenerator(new RandomPath(new VertexCoverage(100)));

    // Get the starting vertex (v_Start)
    setNextElement(model.getVertices().get(0));

    //Create the machine that will control the execution
    Machine machine = new SimpleMachine(this);

    // As long as the stop condition of the path generator is not fulfilled,
    // hasNext will return true.
    while (machine.hasNextStep()) {

        //Execute the next step of the model.
        machine.getNextStep();
    }
}
```

Figure 28. Context a Simple State Machine Model in GraphWalker Test Automation.

Tests for Integration, Functional and Stability: using these tests the test engineer is able to validate end to end, full functionality and stability based on the given time-duration. Configuring the required coverage condition is important to evaluate the applicability of this work. Here, the goal is to attest the constraints using GraphWalker generator algorithms like a

a_star (...) for optimal and shortest path coverage generation with reached vertex or edge keyword stop conditions, and *random (...)* for exhaustive coverage with conditions given in percentage (e.g., 80, 90, 100) and time duration in seconds or milliseconds. These constraints are important to compute the test coverage as stated in Figure 29.

```
...
@Test
public void v_Start() {
    new TestBuilder()
        .addContext(new TestNemoRouteAPISearch().setNextElement(new Edge().setName("e_init").build()),
            MODEL_PATH,
            new AStarPath(new ReachedVertex("v_RestBaseURLPath")))
        .execute();
}

@Test
public void runSmokeTest() {
    new TestBuilder()
        .addContext(new TestNemoRouteAPISearch().setNextElement(new Edge().setName("e_init").build()),
            MODEL_PATH,
            new AStarPath(new ReachedVertex("v_Logout")))
        .execute();
}

@Test
public void runFunctionalTest() {
    new TestBuilder()
        .addContext(new TestNemoRouteAPISearch().setNextElement(new Edge().setName("e_init").build()),
            MODEL_PATH,
            new RandomPath(new EdgeCoverage(100)))
        .execute();
}

@Test
public void runStabilityTest() {
    new TestBuilder()
        .addContext(new TestNemoRouteAPISearch().setNextElement(new Edge().setName("e_init").build()),
            MODEL_PATH,
            new RandomPath(new TimeDuration(30, TimeUnit.SECONDS)))
        .execute();
}
```

Figure 29. Step By Step Test Case GraphWalker Test Automation.

Accordingly, *runSmokeTest* method applies generator *AStarPath* which traverses the *ReachedVertex ()* with condition “v_Logout” vertex in which it starts at the edge element value “e_init” from the model context.

runFunctionalTest method applies generator *RandomPath* which traverses the *EdgeCoverage ()* percentage of 100 with the edge element initialization with “e_init” from the model context. *runStabilityTest* method applies generator

RandomPath which traverses the edge "*e_init*" on the specified time duration of 30 seconds based on the context model path.

7.4. Comparison Analysis

The analysis covers two main results. One is the result obtained from QF-Test and Jenkins QF-Test plugin. Two is the result obtained from step by step GraphWalker implementation and execution. An elaboration for this is as follows:

Results and Discussions for QF-Test and Jenkins QF-Test: test coverage report used in this work shows a major difference especially in response time, and discovers an explicit labeling to the small gray shadow pie-chart in the QF-Test report generation. For this, QF-Test Jenkins plugin has successfully disclosed this gray shadow icon through labeling it as "not executed" on the pie-chart diagram (see

Figure 23). This "not executed" shadow part refers to the time out exception which is not notified during running the test suites node using QF-Test. The table's column names are in shorten forms: *TSuites*: Node of test suites; *TSet*: Test set; *TCase*: Test case, and *Status*: passed or Failed. Table 3 creates data for test suites, test set, test case and status as column names based on the QF-Test test report generation in Figure 21. Each column has row values attached to it. For instance, the column *Tset* contains *URL SetUP* and *Scenario* row values. This table has row value such as *Get route details steps* for column *TCase* to the *TSet* for Scenario row value. All these values mentioned above are evaluated and generated by executing *NEMoRouteDetails.qft* node given to the column *TSuites*. Results in Figure 30 are extracted from the QF-Test and *Jenkins QF-Test*: report generation for making comparisons and discussions (see Figures 22 and Figure 24).

Table 3. QF-Test Test Suites with Test Set, Test Case and Status.

TSuites	TSet	TCase	Status
NEMoRouteDetails.qft	URL SetUP		Passed
	Scenario	Get route details steps	Passed

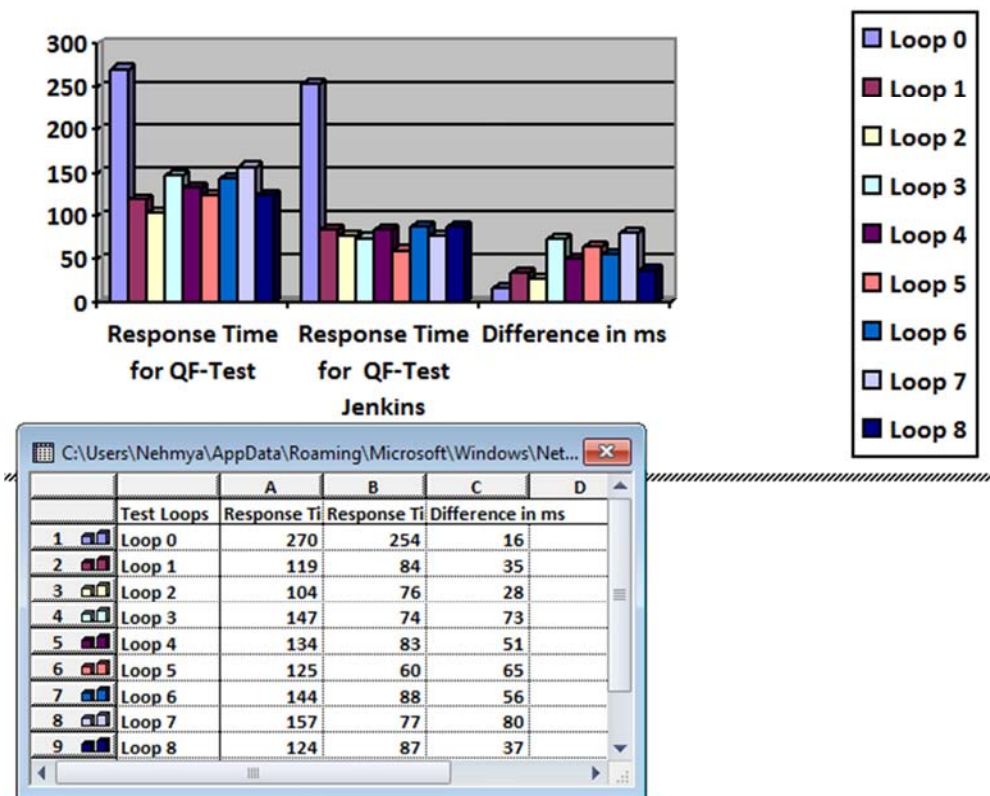


Figure 30. Results Obtained Using Comparison Between QF-Test VS QF-TEST Jenkins.

Difference from Jenkins QF-Test to QF-Test: In Figure 30, the data under first column is taken from Figure 22 and Figure 23, each test step has a significant difference in response time. Example: the total 9 test suites difference between QF-Test and QF-Test Jenkins is 441 ms and its Average Response Time (ART) is $441/9=49$ ms. This 49 ms implies that there is the significant difference in the response

time which is around a delay of 49 extra overhead test cost resource utilization for QF-Test runtime test execution. Moreover, QF-Test takes more test time than Jenkins QF-Test coverage analysis. As indicated by the authors in [5], when this test time increases in service testing, the test becomes so costly and resource intensive. So, QF-Test industry test certification has to be collaborated with testing team for a

Careful design in continuous integration automation framework which improves a software quality test coverage analysis. In such a context, the response time mainly affects the maintainability and reliability quality aspects; however, it can be coped with it by configuring the health amplification factor on the Jenkins QF-Test plugins. These aspects are the predominant factors to impose on quality index of other related dependency quality factors such as security, availability, functionality and usability. Therefore, QF-Test Jenkins integration test report analysis is by far the best way to calculate the optimal test coverage for verifying the quality of services in the cloud.

Outcome for automated test cases: this implementation allows the test engineers examining each FSM as a test model applies the generated keywords based on the Gherkin BDD syntax. Test model ensures the verification process through passing each keyword on a testing framework, TestNG. As described above, the input for this testing is taken from the output of Figure 17. Since this output has passed the validation process through an *optimal A* Search algorithm*. By this, such new approach is a reasonable sound for testing of complex system like a cloud service. Thus, GraphWalker automating test cases enable to use the heuristic approach that allows covering an optimal and efficient route path for a complex mobility system. For instance, *NEMoMobility* Service for tracking transport mode uses the start and end states (nodes) including edges as forwarding inputs for retrieving the data coverage. This ends up the discussion that this work also supports how to use the black-box approach for test coverage (or quality) measurement.

8. Conclusion

This paper begins with motivations that extend to underline the fundamental concepts which intent to devise the black box strategy using a high level specification agile testing for the acceptance test cycle especially an interface end point, that is, REST API. This specification aims to define the problem domain in the context of model-driven engineering (MDE) for enabling the behavior-driven development (BDD). All tasks defined in this paper base from this API that includes the specification model on NEMO mobility service especially in [34]. Thus, the major lessons learned in this work are: describing conceptual idea to elaborate co-existing domain models in software project which binds the concepts available in a PIM and PSM approach, realizing the PIM meta model first to implement the idea of PSM through DSLs with meta programming system (PMS) and/or Xtext template, creating graphic UML for user model which interacts with the DSL Gherkin meta model functional realization, introducing a DSL Gherkin BDD model to generate a GraphWalker digraph model, generating the paths based on keywords through understanding the context finite machine on the defined test path model, defining the test execution to cover the test set up on the scenario based on Gherkin cucumber BDD tests,

reporting test coverage using QF-Test, and Jenkins QF-Test continuous integration and comparing test report analysis available in both QF-Test and Jenkins against prototyping step by step implementation as option to prepare tests for enhanced test coverage. The outcome of this research arises a new effort in software testing paradigm.

The result from this outcome shows a significant resource utilization difference in time spent in tests and expected runtime between QF-Test and QF-Test Jenkins test coverage reports. The difference accounts a test cost of 49 ms delay in QF-Test execution. This number is improved via Jenkins through setting a scaling 1.0% health amplification report which copes with the robustness and reliability failing tests as 99% health by reusing and building the same scenario in QF-Test Jenkins automation testing framework. As a future work, the test engineers should consider other high level approach like a robot framework [17, 35] combined with GraphWalker test path generation for validating the same scenario. In addition to this, an intelligent test coverage like an Evosuite test generation genetic algorithm search-based techniques [40] supplemented with model-based pattern approach [36] should be investigated to attain the optimal stubs running the behaviors generated to validate the cloud service. In such thought, step by step test assertions are useful to measure the test coverage of services as System under Test (SUT).

Acknowledgements

We highly credit constant support from the members of Software Engineering Group, University of Oldenburg. We also appreciate the Quality First Software GmbH, Company, Germany, for their sincere help to offer QF-Test License for conducting this experimental research analysis. We are delightful to forward thanks for DAAD collaboration that supports this research together with Ethiopian-Engineering Capacity Building Program (EECBP). The research leading to these results has received funding from EECBP grant 57251549.

References

- [1] Wolde, Behailu Getachew, and Abiot Sinamo Boltana. "Behavior-Driven Re-engineering for Testing the Cloud." In 2020 Seventh International Conference on Software Defined Systems (SDS), pp. 75-82. IEEE, 2020.
- [2] Ahmed, Anastasia. "Unit test automation with Jenkins-CI tool." (2015).
- [3] "Automate your API tests with Postman," accessed on: Sep. 14, 2019. [Online]. Available: <https://www.postman.com/automatedtesting>, 2019.
- [4] Bashir, Raja Sehrab, Sai Peck Lee, Saif Ur Rehman Khan, Victor Chang, and Shahid Farid. "UML models consistency management: Guidelines for software quality manager." International Journal of Information Management 36, no. 6, 2016.

- [5] Blokland, Kees, Jeroen Mengerink, and Martin Pol. Testing cloud services: how to test SaaS, PaaS & IaaS. Rocky Nook, Inc., 2013.
- [6] Bussenot, Robin, Hervé Leblanc, and Christian Percebois. "Orchestration of Domain Specific Test Languages with a Behavior Driven Development approach." In 2018 13th Annual Conference on System of Systems Engineering (SoSE), pp. 431-437. IEEE, 2018.
- [7] Castanier, Raphaël, and Lars Gustafsson. "How to Test Using Jenkins?."
- [8] Cetinkaya, Deniz, and Alexander Verbraeck. "Metamodeling and model transformations in modeling and simulation." In Proceedings of the 2011 Winter Simulation Conference (WSC), pp. 3043-3053. IEEE, 2011.
- [9] Chelimsky, David, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. "The RSpec book: Behaviour driven development with Rspec." Cucumber, and Friends, Pragmatic Bookshelf 3 (2010): 25.
- [10] Cicchetti, Antonio, Davide Di Ruscio, and Alfonso Pierantonio. "A metamodel independent approach to difference representation." J. Object Technol. 6, no. 9 (2007): 165-185.
- [11] Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans. Foundations of software testing: ISTQB certification. Cengage Learning EMEA, 2008.
- [12] Gutiérrez, Javier J., Isabel Ramos, Manuel Mejías, Carlos Arévalo, Juan M. Sánchez-Begines, and David Lizcano. "Modelling Gherkin Scenarios Using UML." (2017).
- [13] "GraphWalker, Modeling Syntax When Using yEd," accessed on: Nov. 14, 2019. [Online]. Available: <https://graphwalker.github.io/>, 2019.
- [14] Haver, Thomas. "Cucumber 3.0 and Beyond."
- [15] Höfer, C. N., and Georgios Karagiannis. "Cloud computing services: taxonomy and comparison." Journal of Internet Services and Applications 2, no. 2 (2011): 81-94.
- [16] Hoisl, Bernhard, and Stefan Sobernig. "Consistency Rules for UML-based Domain-specific Language Models: A Literature Review." In ACES-MB&WUCOR@ MoDELS, pp. 29-36. 2015.
- [17] Iborra, Andres, Diego Alonso Caceres, Francisco J. Ortiz, Juan Pastor Franco, Pedro Sanchez Palma, and Barbara Alvarez. "Design of service robots." IEEE Robotics & Automation Magazine 16, no. 1 (2009): 24-33.
- [18] Javier Luis C'ánovas Izquierdo and Jordi Cabot, 2014. Composing jsonbased web apis. In International Conference on Web Engineering, pages 390-399. Springer, 2014. [17] Izquierdo, Javier Luis C'ánovas, and Jordi Cabot. "Composing JSON-based web APIs." In International Conference on Web Engineering, pp. 390-399. Springer, Cham, 2014.
- [19] "Converting JSON to POJOs Using Java," accessed on: April 08, 2020. [online]. Available: <https://dzone.com/articles/convertingjson-to-pojos-using-java>, 2020.
- [20] "Generate Plain Old Java Objects from JSON or JSONSchema," accessed on: Aug. 18, 2019. [online]. Available: <http://www.jsonschema2pojo.org/>, 2019.
- [21] Kahani, Nafiseh, and James R. Cordy. "Comparison and evaluation of model transformation tools." Queen's University, Kingston, Tech. Rep. (2015).
- [22] Kozlovics, Sergejs. "Models and Model Transformations Within Web Applications." In International Baltic Conference on Databases and Information Systems, pp. 53-67. Springer, Cham, 2016.
- [23] Kramer, Max Emanuel. Specification languages for preserving consistency between models of different languages. Vol. 24. KIT Scientific Publishing, 2019.
- [24] Kuryazov, Dilshodbek, and Alexander Sandau. "Vertical prototype documentaion." Unpublished, 16 November 2017.
- [25] Kuryazov, Dilshodbek, Andreas Winter, and Alexander Sandau. "Sustainable Software Architecture for NEMO Mobility Platform.", <http://www.se.uni-oldenburg.de/documents/kuryazovWinterSandau2018.pdf>, 2018.
- [26] Leblebici, Erhan. "Inter-Model Consistency Checking and Restoration with Triple Graph Grammars." PhD diss., Technische Universität, 2018.
- [27] Mayer, Thomas, Falk Pappert, Oliver Rose, and Neubiberg Germany München. "A Natural-language-based Simulation Modelling Approach." Simulation in Production and Logistics (2015): 661-670.
- [28] Meacham, Sofia, Vaclav Pech, and Detlef Nauck. "Classification Algorithms Framework (CAF) to Enable Intelligent Systems Using JetBrains MPS Domain-Specific Languages Environment." IEEE Access 8 (2020): 14832-14840.
- [29] Mell, Peter, and Tim Grance. "The NIST definition of cloud computing." (2011).
- [30] "Drawing UML with PlantUML:PlantUML Language Reference Guide, version: 1.2019.9," accessed on: March. 14, 2020. [online]. available: <https://github.com/plantuml>, 2019.
- [31] Pyshkin, Evgeny, Maxim Mozgovoy, and Mikhail Glukhikh. "On requirements for acceptance testing automation tools in behavior driven software development." In Proceedings of the 8th Software Engineering Conference in Russia, 2012.
- [32] "Quality First Software GmbH, Version 4.7.1", accessed on: May 15, 2020. [online]. Available: https://archive.qfs.de/qftest/manual_en.pdf, 2020.
- [33] Rüter, Jan. "A feature model for web testing tools." (2015).
- [34] Alexander Sandau. "NEMO Technical Specification Document: Interface Description NEMO planning System, version: 0.3.2." Unpublished, 27 November 2018.
- [35] Schlegel, Christian, Andreas Steck, and Alex Lotz. "Model-driven software development in robotics: Communication patterns as key for a robotics component model." Introduction to Modern Robotics (2011): 119-150.
- [36] Sharma, Tushar, Pratibha Mishra, and Rohit Tiwari. "Designite: A software design quality assessment tool." In Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, pp. 1-4. 2016.
- [37] Silva, Thiago Rocha. "Definition of a behavior-driven model for requirements specification and testing of interactive systems." In 2016 IEEE 24th International Requirements Engineering Conference (RE), pp. 444-449. IEEE, 2016.

- [38] Sivanandan, Sandeep. "Agile development cycle: Approach to design an effective Model Based Testing with Behaviour driven automation framework." In 20th Annual International Conference on Advanced Computing and Communications (ADCOM), pp. 22-25. IEEE, 2014.
- [39] Sneed, Harry M., 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), pp. 59-66. IEEE, 2015.
- [40] Vivanti, Mattia, Andre Mis, Alessandra Gorla, and Gordon Fraser. "Search-based data-flow test generation." In 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 370-379. IEEE, 2013.
- [41] Volter, Markus. "From programming to modeling-and back again." IEEE software 28, no. 6 (2011): 20-25.
- [42] Wolde, Behailu Getachew, and Abiot Sinamo Boltana. "Combinatorial Testing Approach for Cloud Mobility Service." In Proceedings of the 2019 2nd Artificial Intelligence and Cloud Computing Conference, pp. 6-13. 2019.
- [43] Yau, Stephen, and Ho An. "Software engineering meets services and cloud computing." Computer 44, no. 10 (2011): 47-53.
- [44] Russell, Stuart, and Peter Norvig. "Artificial intelligence: a modern approach." (2002).