2016 IEEE International Conference on Software Testing, Verification and Validation

Skyfire: Model-Based Testing With Cucumber

Nan Li, Anthony Escalona, and Tariq Kamal
Research and Development
Medidata Solutions
{nli, aescalona, tkamal}@mdsol.com

***Abstract*—In the software industry, a Behavior-Driven Development (BDD) tool, *Cucumber*, has been widely used by practitioners. Usually product analysts, developers, and testers manually write BDD test scenarios that describe system behaviors. Testers write implementation for the BDD scenarios by hand and execute the *Cucumber* tests. *Cucumber* provides transparency about what test scenarios are covered and how the test scenarios are mapped to executable tests. One drawback of the *Cucumber* BDD approach is that test scenarios are generated manually. Thus, the test scenarios are usually weak. More importantly, practitioners do not have a metric to measure test coverage.**

In this paper, we present a Model-Based Testing (MBT) tool, *skyfire*. *Skyfire* can automatically generate effective *Cucumber* test scenarios to replace manually generated test scenarios. *Skyfire* reads a behavioral UML diagram (e.g., a state machine diagram), identifies all necessary elements (e.g., transitions) of the diagram, generates effective tests to satisfy various graph coverage criteria, and converts the tests into *Cucumber* scenarios. Then testers write *Cucumber* mappings for the generated scenarios. *Skyfire* does not only generate effective tests but is also completely compatible with the existing agile development and continuous integration (CI) rhythm. We present the design architecture and implementation of *skyfire*, as well as an industrial case study to show how *skyfire* is used in practice.

I. Introduction and Motivation

In the software industry, practitioners have widely adopted a Behavior-Driven Development (BDD) approach that uses *Cucumber* [1]. Under this methodology, product analysts, developers, and testers manually create BDD scenarios that describe behaviors of a System Under Test (SUT). Each test scenario consists of several steps written in human languages (e.g., English), with *Cucumber* syntax. Testers write test code for each step of the test scenarios, and then run the *Cucumber* tests which generate test reports. Note that the BDD approach is used in high-level testing such as system testing or User Acceptance Testing (UAT), but not in low-level testing such as unit testing. In industry, the distinction among integration testing, system testing, and UAT often blurs. To concentrate on system behaviors from **users**' perspective, we use UAT to refer to the level at which the BDD approach is used.

The major benefit of using *Cucumber* is to provide transparency among product analysts, developers and testers. Product analysts who are generally non-technical people, can easily understand what system behaviors BDD scenarios cover. *Cucumber* provides a mapping mechanism so that testers can write test code to map to each distinct BDD scenario step. Thus, it is also easy for everyone to understand whether tests are correctly implemented. Although *Cucumber* has been widely used in industry, it has three negative drawbacks when BDD test scenarios are generated manually. First, the test scenarios are usually weak and do not cover complicated cases. Second, practitioners do not have a metric to measure the BDD tests. Third, when a system behavior changes, practitioners have to manually change all occurrences related to that behavior in the test scenarios. So practitioners are looking forward to an approach that can automatically generate test scenarios.

On the other hand in academia, *Model-Based Testing (MBT)* has been used to generate tests automatically from behavioral models. Researchers have devised many MBT methodologies and techniques. We are interested in whether we could leverage MBT to create test scenarios for *Cucumber*. We are inspired to use MBT with *Cucumber* because MBT uses test models that describe system behaviors. This is in alignment with the BDD approach at the UAT level in industry. In addition, we discover that practitioners often use behavioral models because visual diagrams are easier for people to understand requirements and system behaviors. However, many practitioners have not leveraged their behavioral models to generate tests using MBT. The main reason is that prior MBT techniques are complicated to use and they are often not compatible with existing tools in agile development. Prior MBT techniques usually create complicated models that contain lots of additional information including inputs and mappings and provide intuitive mapping mechanisms from model to test implementation. This would force practitioners to give up what they have been using in practice and take a completely new approach. Therefore, the question of how to adapt MBT techniques into existing industrial agile development environment is of significant importance.

In this paper, we present a MBT tool, *skyfire* [2], which is seamlessly integrated with the existing BDD approach using *Cucumber*. *Skyfire* reads a behavioral UML diagram (e.g., a state machine diagram), identifies all necessary elements, and generates effective test scenarios to satisfy a graph coverage criterion. The generated tests consist of identifiable elements (e.g., transitions and constraints) from the diagram. *Skyfire* converts these tests into *Cucumber* scenarios. Then practitioners leverage the *Cucumber* mapping mechanism to create mappings from model to concrete tests. To the best of our knowledge, this is the first time MBT is used to generate test scenarios for *Cucumber*.

With this new approach, practitioners only need to generate test scenarios from a UML behavioral diagram using *skyfire*. With the mapping mechanism of *Cucumber*, practitioners can

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.41

fully concentrate on model building to capture key system behaviors and not worry about how to attach mapping and input information to the model. The approach that uses *skyfire* and *Cucumber* will not only generate effective tests, but also not hinder existing agile development and continuous integration (CI) rhythm. CI is an agile development practice in which every code commit is pulled automatically from a source code repository and verified automatically by an automated build. Because practitioners can use *skyfire* to develop tests very easily at the same directory as SUT, *skyfire* is completely integrated with existing build and CI tools. We will show how to apply *skyfire* to a product at Medidata.

The paper is organized as follows. Section II introduces the BDD methodology that uses *Cucumber*. Section III presents the background of the MBT techniques. Section IV shows the detailed architecture and implementation of *skyfire*. Section V gives an industrial case study to which we have applied *skyfire* and *Cucumber*. Section VI discusses related work about generating *Cucumber* test scenarios. Section VII concludes this research and talks about future work.

II. BDD Methodology of Cucumber

Cucumber is a testing tool that supports behavior-driven development. This tool has been widely used in the software industry. Practitioners use *Cucumber* to generate and run UAT tests in three steps. First, product analysts, developers, and testers work together to write *Cucumber* scenarios in feature files. A feature file usually contains a list of scenarios. A *Cucumber* scenario consists of steps, known as *Given*s, *When*s, and *Then*s. *Given* steps declare the system is in a known state. *When* steps show the users' actions. *Then* steps verify the system outputs after the actions. *Cucumber* also supports *Add* steps to connect multiple *Given*, *When*, and *Then* steps to make the writing more fluent. Second, testers write test code for every distinct step in the test scenarios using the *Cucumber* mapping mechanism, called *step definitions*. Third, testers run the tests and get test reports using *Cucumber* locally and in a CI environment.

We use a real-world product at Medidata, *Roc*, as an example to show how we use *Cucumber* to generate tests. Later in Section V, we will discuss how to use *skyfire* and *Cucumber* to generate MBT tests for *Roc*. *Roc* has been developed in Java. It is a service that wraps the *Elastic MapReduce (EMR)* of *Amazon Web Services (AWS)*. *EMR* is a web service that processes large amounts of data efficiently using *Hadoop*[1]. *Roc* serves as an infrastructure for big data applications, providing easier APIs for developers to use *EMR* to process high volumes of data and monitor the process. For example, by using *Roc*, developers can easily start an *EMR* cluster, add extra steps to execute, get the *EMR* status, terminate an *EMR* cluster, etc. At the UAT level, we need to design test scenarios for *Roc*, along with product analysts and developers. The example below shows a test scenario generated manually. This scenario creates an EMR cluster and terminates the EMR cluster for *Roc* users.

Features: *EMR-related operations*
Scenario: *Create an EMR cluster with a valid credential*
Given *a user with a valid credential*
When *the user starts an EMR*
Then *the user should see a success status returned by the EMR*
When *the user terminates the EMR*
Then *the user sees a success status from the EMR*

After all the scenarios are generated, testers write test code to map to the scenario steps in step definitions. The example below shows partial implementation. The annotation above each method is mapped to the steps in the test scenarios.

```
public class EmrStepDefinitions {
  private RocClient rc;
  private CreateEmrRequest createRequest;
  private CreateEmrResponse createResponse;
  ...
  Given ("∧ a user with a valid credential $")
  public void a_user_with_a_valid_credential() {
    ...
    rc = new RocClient (privateKey, publicKey, ...);
    ...
  }

  When ("∧ the user starts an EMR $")
  public void the_user_starts_an_EMR() {
    ...
    createRequest = new CreateEmrRequest();
    createResponse = rc.postRequest(createRequest);
    ...
  }

  Then ("∧ the user sees a success status from the EMR $")
  public void the_user_starts_an_EMR() {
    Assert.assertEquals("SUCCESS", createResponse.getStatus());
  }
}
```

When step definitions are created, testers use annotations to specify the feature files to execute in a test. Then *Cucumber* will look for the step definitions, execute the test scenarios, and generate test reports.

```
@RunWith (Cucumber.class)
@Cucumber.Options (features = "pathToFeatureFile")
```

From the examples above, we can see that *Cucumber* provides clear transparency among products staff, developers, and testers. Product analysts can understand scenarios easily since the scenarios are almost written in plain English. The mappings from scenario steps to test code are also relatively clear. At some companies, tests are subject to auditing so it is important for non-technical people including product and auditing staff to understand them. For example, all Medidata products are subject to audits from the Food and Drug Administration (FDA).

From a technical perspective, the mapping mechanism is smart because testers only need to create mappings for distinct steps but not all steps in test scenarios. A distinct step may appear more than once in test scenarios. Thus, creating mappings for distinct steps saves time and effort. This idea

[1]Apache Hadoop processes large data sets over clusters of computers using *Hadoop Distributed File System (HDFS)*.

is similar to the mapping strategy used in many prior MBT techniques. Moreover, the test scenarios are separated from implementation. So the scenarios are programming language-independent. *Cucumber* supports almost all kinds of programming languages. For the same test scenarios, testers can easily create mappings in different languages.

As mentioned in Section I, the drawback of the existing approach using *Cucumber* is obvious. The effectiveness of tests depends on test scenarios, which, however, are created manually and therefore weak. More importantly, unlike unit testing that has statement coverage, branch coverage, etc., there are no coverage criteria for testers to measure the quality of BDD tests. Practitioners are looking forward to using coverage criteria to generate effective test scenarios.

III. Model-Based Testing

In *model-based testing* (*MBT*), testers generate tests from behavioral models that reflect the functional aspects of a system. For example, Finite State Machines (FSMs) that represent the behavior of a system are often used to generate tests. Figure 1 shows a general process to derive tests from behavioral models, which is explained in detail in the next paragraph.

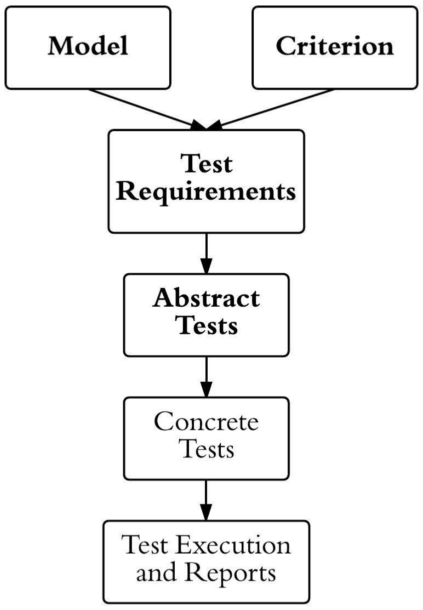

Fig. 1. A Generic Model-based Testing Process

Testers can generate test requirements by hand or by using a coverage criterion based on a model. Because using coverage generates effective tests and tells testers when to stop testing, we choose to use coverage criteria to generate tests. *Test requirements* are specific elements of software artifacts that must be satisfied or covered. An example test requirement for the test criterion "cover every node" is "cover the initial node." A *test criterion* is a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests. *Abstract tests*, which are expressed in terms of a model, are generated to satisfy the test requirements. *Concrete tests* are expressed in terms of the implementation of the model, and are ready to be run automatically. Testers usually need to create *mapping*s to transform abstract tests to concrete tests. Expected outputs can be specified in models or provided manually by testers.

While creating test models, testers can either write *model program*s in a specific language or draw visual diagrams. Model programs use specification languages such as Spec# in Spec Explorer [3] or programming languages such as C# in NModel [4] to describe model behaviors. Then the model programs can be converted to finite state machines or extended finite state machines. More testers use visual diagrams such as UML behavioral diagrams directly as test models rather than model programs. UML behavioral models are widely used and easy to understand. At Medidata, we have used UML state machine diagrams to describe system behaviors. Furthermore, we will use the diagrams to generate tests.

We summarize four problems that prevent practitioners from adopting some of the previous MBT techniques. First, previous MBT techniques often use complicated approaches with multiple models. Models usually include lots of additional information used for transforming abstract tests to concrete tests. For example, Briand and Labiche [5] used supporting UML models (e.g., use case diagrams and class diagrams) to provide the additional information in their TOTEM system, which assumes the existence of artifacts that are not always available. Nebut et al. [6] presented a use case driven approach, which extracted additional information from use case models and required a behavioral model (sequence, state machine, or activity diagram) to specify the sequence ordering of the use cases. They assumed use cases had contracts to help infer the partial ordering of functionalities. Furthermore, the behavioral models had to be consistent with the use cases. That is, the parameters in use cases were assumed to be the same as those in the behavioral models. This makes model creation extremely difficult, distracting practitioners from focusing on system behaviors. Second, previous MBT techniques have their own mapping mechanisms from abstract tests to concrete tests, with complicated models. This requires a steep learning curve to master the new mapping methods. Third, previous MBT techniques usually provide language / platform-specific solutions when generating concrete tests. Adopting these MBT techniques into a different language or platform requires a great effort, making practitioners reluctant to use them. Fourth, previous MBT research tools may not be compatible with industrial agile development and CI environments.

To overcome the problems above, we want to build a tool (*skyfire*) that is able to 1) generate tests from a single UML diagram under the SUT, 2) leverage the mapping mechanism and multi-language / platform support of *Cucumber*, and 3) be compatible with agile CI environments. Therefore, *skyfire* creates *Cucumber*-compatible tests from behavioral models.

The MBT processes covered by *skyfire* are in bold in Figure 1. In Section IV, we will discuss the architecture and implementation of *skyfire*.

IV. Skyfire

We developed a MBT tool *skyfire* based on our previous research, a Structured Test Automation Language framEwork (STALE) [7], [8]. STALE reads UML behavioral models, generates abstract tests, provides a test automation language for users to create mappings from the model to test code, and generates concrete tests based on the mappings. STALE also provides a graphical interface for users to create mappings easily. STALE requires users to put the UML model and source code of the SUT under a directory of STALE. Since we can leverage *Cucumber* to create mappings from abstract tests to concrete tests, *skyfire* only needs to reuse the model reading and abstract test generation from STALE. In addition, *skyfire* needs to convert abstract tests to *Cucumber* test scenarios. We started developing *skyfire* by cloning STALE. Then we made the following changes to *skyfire* in order to integrate it with *Cucumber* BDD methodology and make it compatible with our agile development and CI environment.

1) Removed the concrete test generation from STALE since we use *Cucumber* to create mappings and generate concrete tests
2) Added the transformation from abstract tests to *Cucumber* test scenarios
3) Added *Apache Maven* build automation tool [9] to meet the industrial standard of agile development
4) Added public APIs so that users can generate *Cucumber* test scenarios at the repository of SUT
5) Added the configuration file of a CI tool, *Travis* [10], to *skyfire* so that *skyfire* is built automatically for every single code commit
6) Added a code coverage tool, *JaCoCo* [11], to measure statement and branch coverage
7) Added a static analysis tool, CheckStyle [12], to improve the code quality
8) Added a logging tool *Apache Log4j*
9) Uploaded the 1.0.0 version to the Maven central repository. Users can download *skyfire* through Maven.

Figure 2 shows the architecture of *skyfire*. *Skyfire* consists of five major modules: Eclipse Modeling Framework (EMF) [13] model reader, general graph converter, test path generator, abstract test generator, and *Cucumber* scenario Transformer. *Skyfire* reads EMF-based UML behavioral models and identifies model elements. EMF has a large community and supports various open source modeling tools. Currently *skyfire* supports EMF-based UML state machine diagrams. *Skyfire* converts a UML state machine diagram to a general graph, which is formally defined in Section IV-A. With a specified graph coverage criterion, *skyfire* generates test paths that consist of graph nodes and edges. *Skyfire* then generates abstract tests that consist of states, transitions, and constraints, by mapping the test paths to the original EMF-based UML state machine diagram. Finally, *skyfire* converts the abstract tests to *Cucumber* test scenarios.

We have applied several best practices of industrial Java development to *skyfire*. First, we have used *Apache Maven* [9] to build *skyfire*. *Maven* is a widely used build automation tool for Java projects. *Maven* provides an excellent mechanism to manage dependencies and add plugins in a *Project Object Model (POM)* file. To improve the code quality of *skyfire*, we have used JaCoCo [11] to measure the statement and branch coverage for *skyfire*. To avoid potential faults and improve readability, we have applied CheckStyle [12], a static analysis tool. *Skyfire* [2] is publicly available on Github, the most popular cloud-based source hosting repository. *Skyfire* is open sourced under the MIT license.

One of the best practices of agile development and testing is to have a CI environment. We chose to use *Travis* [10] as the CI tool for *skyfire*. *Travis* CI is cloud-based, providing a continuous software integration service to developers and currently running in thousands of open source projects. By default, Travis builds project files using the *.travis.yml* file. This file is added to the *skyfire* Github repository and triggers a *Travis* CI build each time a new commit has been made to *skyfire*. *Travis* executes the *Maven* build for *skyfire*.

Section IV-A defines general graphs to which UML diagrams are converted and four graph coverage criteria that *skyfire* uses. Section IV-B describes the transformation rules from UML state machine diagrams to general graphs and algorithms used to generate test paths. Section IV-C gives the transformation rules from abstract tests to *Cucumber* test scenarios. Section IV-D presents how to use *skyfire*.

A. Graph Coverage Criteria

Skyfire uses graph coverage criteria, which are defined on general graphs. The following definitions are taken from Ammann and Offutt's software testing book [14]. A graph G is drawn as a collection of circles connected by arrows, where the circles represent nodes and the arrows represent edges. Formally, G is

- a set N of *nodes*, where $N \neq \emptyset$
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
- a set N_f of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$
- a set E of *edges*, where E is a subset of $N \times N$

Thus, a test graph G **requires** at least one initial and one final node, but **allows** more initial and final nodes.

Edges are considered to be directed arcs *from* one node n_i and *to* another n_j and written as (n_i, n_j). A *path* is a sequence $[n_1,\ n_2,\ ...,\ n_M]$ of nodes, where each pair of adjacent nodes, $(n_i,\ n_{i+1})$, $1 \leq i < M$, is in the set E of edges. The *length* of a path is defined as the number of nodes it contains. A *subpath* of a path p is a subsequence of p (including p itself). A *test path* represents the execution of a test case on a graph. Test paths must start at an initial node and end at a final node. A test path p *covers* a subpath q if q is a subpath of p.

Below we formally define the four graph coverage criteria supported by *skyfire*. *Node coverage* results in one test require-

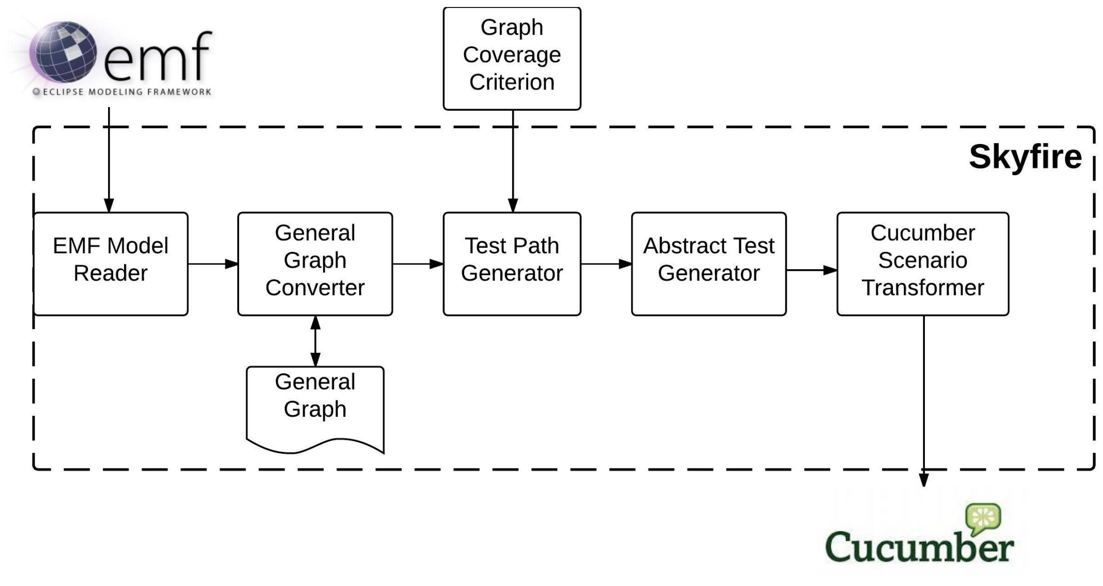

Fig. 2. The Architecture of skyfire

ment for each node in the graph. *Edge coverage* requires that each edge is covered by test paths.

*Definition 1 (**Node Coverage**):* TR contains each node, inclusive, in G.

*Definition 2 (**Edge Coverage**):* TR contains each edge, inclusive, in G.

The test requirements of EPC are subpaths of length no more than three (the length is measured in terms of the numbers of nodes) in G. Thus, *edge-pair coverage* subsumes edge coverage (test requirements of length of two) and node coverage (test requirements of length of one) in G.

*Definition 3 (**Edge-pair Coverage**):* TR contains each reachable path of length up to 2, inclusive, in G.

A path from node n_i to n_j is *simple* if no node appears more than once in the path. That is, simple paths cannot have internal loops. However, if the entire path is a loop that has identical first and last nodes, the loop is a simple path. Graphs may have lots of simple paths, many of which are subpaths of other simple paths. To eliminate the redundancy and maintain the coverage strength, *prime path coverage* was invented [14]. A path from n_i to n_j is *prime* if it is simple and does not appear as a proper subpath of any other simple path.

*Definition 4 (**Prime Path Coverage**):* TR tours each reachable prime path in G.

B. Abstract Test Generation

Skyfire transforms each UML state machine diagram to a general graph with initial nodes and final nodes. For a UML state machine diagram, an initial state is mapped to an initial node in the graph, a final state to a final node, and other states to unique nodes. Each transition becomes an edge and an internal transition from one state to itself becomes a self-loop on the corresponding node. Elements including composite state, choice, fork, junction, and join need special treatment. Each sub-state of a composite state becomes a unique node and the composite state itself will not be transformed to a node. If a composite state has an initial state, an edge will be created for an incoming transition to the initial state of the composite state. If a composite state has n regular sub-states but no initial states, n edges will be created for an incoming transition, one for each node. Likewise, if a composite state has a final state, an edge will be created for an outgoing transition from the final state of the composite state. If a composite state has n sub-states but no final states, n edges will be created for an outgoing transition. An edge will be created for each outgoing transition of a choice or fork. An edge will be created for each incoming transition of a join or junction.

Once a UML state machine diagram is transformed to a general graph, test paths can be generated based on a graph coverage criterion. The prefix graph-based algorithm from Li et al. [15] are used to generate test paths. Some complex graph coverage criterion may generate complicated test requirements such as prime path coverage. The prime paths generated usually have overlapped nodes or sub-paths. The prefix graph-based algorithm generates a coverage-adequate test sets that have fewer numbers of nodes and numbers of total paths. Next, *skyfire* translates the test paths to abstract tests by mapping each node and edge from the test paths to constraints and transitions from the state machine diagram.

C. Transformation to Cucumber Features

The transformation rules from abstract tests to *Cucumber* test scenarios are listed below.

1) A set of abstract tests derived from a single behavioral model is included in a *Cucumber* feature file.

2) Each abstract test is converted to a *Cucumber* test scenario.
3) The first transition in an abstract test is converted to a *Cucumber Given* step because the first transition is used to initialize the system. The name of the step is the transition name.
4) The second transition is converted to a *Cucumber When* step. The name of the step is the transition name.
5) Every constraint is converted to a *Cucumber Then* step. The name of the step is the constraint name.
6) If prior transitions of a transition have been converted to *Cucumber When* or *And* steps, this transition is converted to a *And* step to make the scenario writing more fluent. The name of the step is the transition name.
7) If prior constraints of a constraint have been converted to *Cucumber Then* or *And* steps, this constraint is converted to a *And* step to make the scenario writing more fluent. The name of the step is the constraint name.

If we have an abstract test [*initializeWithValidKeys*, *startEmr*, *emrCreationSuccess*, *addValidSteps*, *terminateEmr*, *emrTerminationSuccess*]. *initializeWithValidKeys*, *startEmr*, *addValidSteps*, and *terminateEmr* are transitions and *emrCreationSuccess* and *emrTerminationSuccess* are constraints. Then the transformed *Cucumber* scenario will look like the one below. The name of a *Cucumber* test scenario is constructed by using all the transition and constraint names from the corresponding abstract test. A *Cucumber* step is described using either a transition or constraint name from the abstract test. If we use clear and meaningful transition and constraint names, the *Cucumber* test scenarios are still easy to understand and provide transparency for product analysts, developers and testers, as opposed to manual test scenarios written in English.

```
Scenario: initializeWithValidKeys startEmr emrCreationSuccess
addValidSteps terminateEmr emrTerminationSuccess
Given: initializeWithValidKeys
When: startEmr
Then: emrCreationSuccess
When: addValidSteps
And: terminateEmr
Then: emrTerminationSuccess
```

We specify a graph coverage criterion when generating test scenarios. The generated Cucumber test scenarios are guaranteed to satisfy this coverage criterion (100% coverage). However, testers may not implement some of the scenarios due to a tight time budget. The final coverage is measured with a ratio of the implemented test scenarios over all test scenarios.

D. Usage

This section presents how to use *skyfire* from users' perspective. First, users need to develop a UML state machine diagram to describe system behaviors for SUT. If users use *Eclipse*, they can use *Papyrus* [16] since *Papyrus* is integrated with *Eclipse*. Other users can use *UMLDesigner* [17], which has a standalone version available for Windows, Linux, and MacOS. Both tools support the latest EMF-based UML standard (UML 2.5.0) with the *.uml* extension.

After having a UML state machine diagram, users need to use *skyfire* to generate *Cucumber* test scenarios from the diagram. If a SUT is developed in Java using *Maven*, users can directly specify the *skyfire* dependency in the *Maven POM* file, as shown below. If users develop a SUT using a non-Java language, they need to create a new *Maven*-based Java project.

```
<dependency>
  <groupId>com.mdsol</groupId>
  <artifactId>skyfire</artifactId>
  <version>1.0.0</version>
</dependency>
```

Next, users only need to make one method call to a public API of *skyfire*, which is the method *generateCucumberScenario* of class *CucumberTestGenerator* shown below. Users need to specify four parameters, the path to the UML behavioral model to use, a graph coverage criterion, the description of the feature file, and the path to the generated feature file. When the feature file is generated, testers will use *Cucumber* to create mappings using the programming language that SUT uses.

```
CucumberTestGenerator.generateCucumberScenario (
  Paths.get (pathToModel),
  TestCoverageCriteria.SOMECOVERAGE,
  featureDescription,
  Paths.get (pathToFeatureFile));
);
```

V. Case Study

We have used *skyfire* to test one of our products at Medidata, *Roc*, mentioned in Section II. The original UAT tests of the 2015 versions were developed manually. In this case study, we developed new model-based UAT tests using *skyfire* and *Cucumber* for *Roc* version 2015.3.0. Table I shows the program details for *Roc* 2015.3.0, measured by a line counter, *Cloc* version 1.6.2 [18].

```
Feature: Roc feature file generated from a state machine diagram

Scenario: initializeWithValidKeys startEmr addValidSteps
          addInvalidSteps checkUntilGettingErrors
......

Scenario: initializeWithValidKeys startEmr startUntilRunning
          addValidSteps checkStepsUntilComplete addValidSteps
          checkUntilRunning runUntilWaiting addInvalidSteps
          checkUntilGettingErrors
Given initializeWithValidKeys
When startEmr
Then emrCreationIsSuccess
When startUntilRunning
Then emrIsRunning
When addValidSteps
And checkStepsUntilComplete
Then stepsAreCompleteWaiting
When addValidSteps
And checkUntilRunning
Then emrIsRunning
When runUntilWaiting
Then stepsAreCompleteWaiting
When addInvalidSteps
And checkUntilGettingErrors
Then invalidStepErrorOccur

Scenario: .....
```

Following the approach we mentioned in Sections III and IV, we designed a UML state machine diagram to capture key

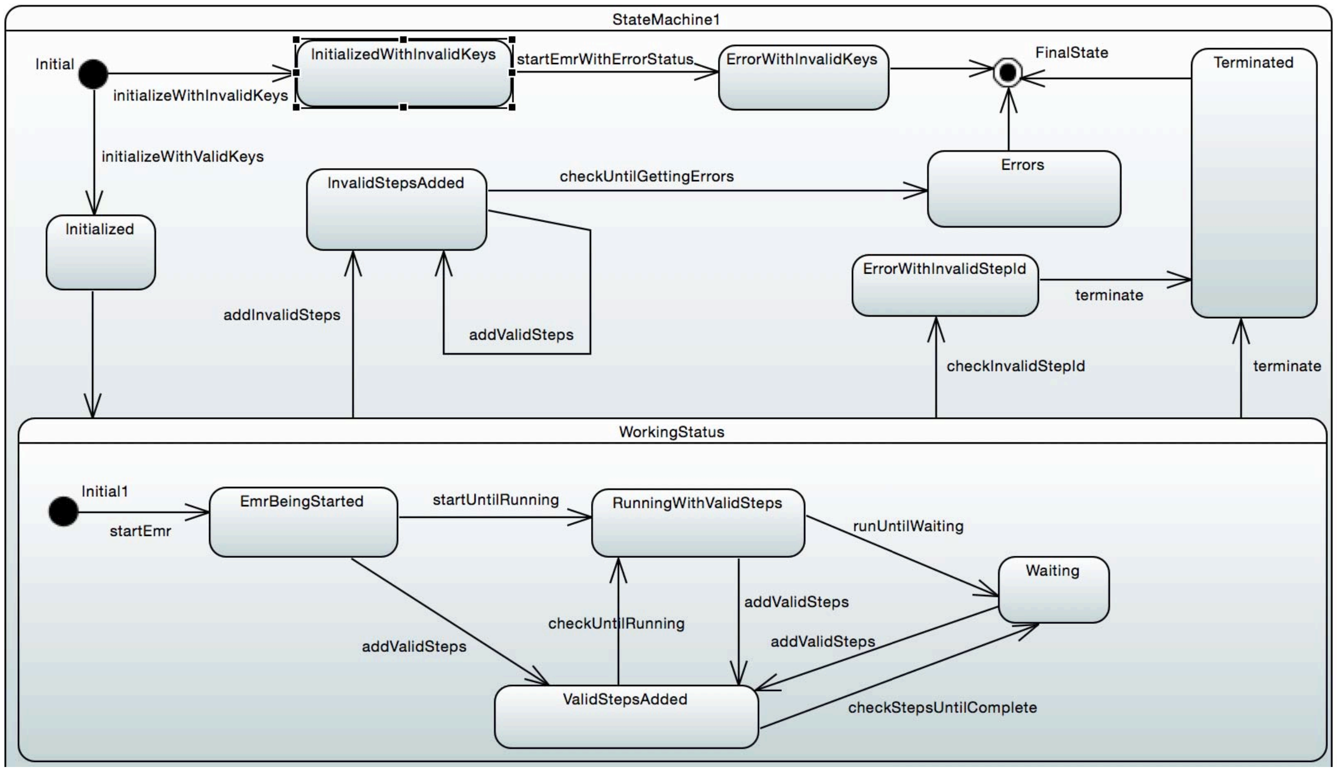

Fig. 3. The State Machine Diagram for *Roc*

TABLE I
THE *Cloc* REPORT FOR *Roc*

Language	#Files	#LOC
Java	137	12,841
XML	28	1,498
Maven	61	699
JSON	10	531
Shell	10	277
Others	4	64
Total	**196**	**15,910**

behaviors of *roc*, as shown in Figure 3. The transitions in the diagram represent the key functions of *roc*. The states represent the statuses returned from *EMR* clusters and data processes. Constraints (state invariants) are included in the states but we cannot see the constraints in Figure 3. The starting state is called "Initial" and the ending state is called "FinalState." The names of all states start with an upper case letter and those of all transitions start with a lower case letter. "WorkingStatus" is a composite state, whose starting state is called "Initial1".

The state machine diagram is converted to a general graph and the graph has 11 nodes and 27 edges. We used edge coverage to generate 13 abstract tests and convert the abstract tests into *Cucumber* test scenarios. The example on page 6 only shows the second test scenario in the feature file. Each abstract test is converted to a test scenario. The name of a scenario is composed of the transition names of the corresponding abstract test. As mentioned in Section IV-C, the first transition that initializes the system is converted to a *Given* step to put the system in a known state. The following transitions are converted into *When* steps to describe key actions. Constraints in states are converted to *Then* steps to verify outcomes.

Once the test scenarios are generated, we developed the step definitions / mappings for each step in the test scenarios. *Cucumber* generates a HTML report after the tests are executed. The dark green color indicates the tests that passed. Otherwise, *Cucumber* marks failed steps with red.

By comparing with the test scenarios created manually, we found the tests generated by *skyfire* to be better in two aspects. First, tests are more effective in terms of the number of actions included in test scenarios. A manual test scenario includes no more than five actions. In contrast, the most complicated *skyfire*-generated scenario (the second scenario in the above feature file) has nine actions. This test could better simulate users' behaviors in a real development environment where *Roc* is used to process complex *EMR* jobs. Second, unlike manual-written steps, *skyfire* generates clear steps for practitioners to map. When checking step definitions / mappings for the manual-written scenarios, we found that some step definitions mis-mapped because some of the steps are long and look similar.

Overall, the MBT approach helps us to develop move effective tests. Previously, when we design test scenarios manually, we are likely to generate simple test cases. For example, we perform one action such as starting an EMR, and then evaluate the output. Now the MBT approach forces practitioners to think harder about the interactions among the system actions. Although designing a state machine diagram may be complicated than the manual test generation, this approach enables practitioners to fully understand the requirements since they now have to identify system states and understand how each action affects other actions and states.

This MBT approach using *skyfire* is compatible with our agile development. Unlike other MBT approaches that usually require practitioners to use a new framework, learn a new language, or maintain complex models, *skyfire* leverages the *Cucumber* BDD methodology used in industry. With *Cucumber*, the concern of mapping model to concrete tests is separated from the MBT approach. When designing models, practitioners only need to focus on system behaviors, not on what input value or mapping information needs to be added to the model. Since we generate tests under the *Roc* repository using *skyfire*, the tests, along with the whole source repository is built automatically under our CI environment that uses another tool Jenkins [19]. *Jenkins* can be used for proprietary projects and can be installed on a local computer. Thus, this approach does not hinder our CI rhythm.

Skyfire is compatible with software evolution and does not add more side-effects than the manual test scenario creation. If system behaviors remain the same but only the APIs are changed, practitioners only need to update mappings as before. If one system behavior is changed, practitioners need only to change the model at one place, not the repeated occurrences of that behavior in the test scenarios as before. Then the test scenarios are automatically generated immediately to reflect the latest change. This is definitely better than the manual approach.

VI. Related Work

Diepenbeck et al. [20] designed an intuitive algorithm to generate *Cucumber* test scenarios to reach 100% method, statement, and branch coverage. They generated additional test scenarios from existing test scenarios. In contrast, we generate *Cucumber* test scenarios using MBT techniques from behavioral models that describe system behaviors independently.

VII. Conclusions and Future Work

In this paper, we present a Model-Based Testing (MBT) tool, *skyfire*. With leveraging the mapping mechanism of *Cucumber*, *skyfire* generates *Cucumber* test scenarios used at the User Acceptance Testing (UAT) level. To the best of our knowledge, this is the first time MBT is used to generate *Cucumber* test scenarios. Practitioners can easily use *skyfire* to generate effective UAT test scenarios from UML state machine diagrams. Four graph coverage criteria supported by *skyfire* provide metrics for practitioners to measure the quality of UAT tests. More importantly, using *skyfire* does not hinder agile development and is completely compatible with the CI rhythm. Considering the wide acceptance of *Cucumber* in the software industry, effectiveness of tests generated by *skyfire*, and usage easiness of *skyfire*, *skyfire* has potential to be widely used to generate BDD test scenarios.

In the future, we will continuously develop *skyfire*. Our recent development plan includes accepting more UML behavioral diagrams besides state machine diagrams, upgrading to the latest UML standard 2.5.0, and fixing several known issues. Other developers are also welcome to contribute since *skyfire* is an open source project.

Acknowledgement

We offer our sincerest gratitude to Isaac Wong, Angela Sloan, and Haider Riaz Khan for supporting this research and reviewing the paper.

References

[1] A. Hellessoy, "Cucumber," Online, 2008, https://cucumber.io/, last access Dec 2015.

[2] N. Li, "Skyfire," Online, 2015, https://github.com/mdsol/skyfire, last access Dec 2015.

[3] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, and N. Tillmann, "Microsoft SpecExplorer," Online, 2002, http://research.microsoft.com/en-us/projects/specexplorer/, last access April 2013.

[4] J. Jacky and M. Veanes, "NModel," Online, 2006, http://nmodel.codeplex.com/, last access April 2013.

[5] L. Briand and Y. Labiche, "A UML-based approach to system testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, ser. UML '99. London, UK: Springer-Verlag, 2001, pp. 194–208.

[6] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Automatic test generation: a use case driven approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 3, pp. 140–155, March 2006.

[7] N. Li and J. Offutt, "A test automation language framwork for behavioral models," in *The 11th Workshop on Advances in Model Based Testing*, ser. A-MOST 2015, Graz, Austria, April 2015.

[8] N. Li, "The structured test automation language framework," Online, 2013, http://cs.gmu.edu/ nli1/stale/, last access Dec 2015.

[9] A. S. Foundation, "Apache maven," Online, 2004, https://maven.apache.org/, last access Dec 2015.

[10] T. C. community, "Travis-CI," Online, 2011, https://travis-ci.org/, last access Dec 2015.

[11] M. Hoffmann, B. Janiczak, E. Mandrikov, and M. Friedenhagen, "Jacoco code coverage tool," Online, 2009, http://eclemma.org/jacoco/, last access Dec 2015.

[12] O. Burn, "Checkstyle," Online, 2001, http://checkstyle.sourceforge.net/, last access Dec 2015.

[13] E. Foundation, "Eclipse modeling framework," Online, 2008, http://www.eclipse.org/modeling/emf/, last access Dec 2015.

[14] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008.

[15] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Proceedings of IEEE 5th International Conference on Software Testing, Verification and Validation*, ser. ICST'12, Montreal, Quebec, April 2012.

[16] E. Foundation, "Papyrus," Online, 2008, www.eclipse.org/papyrus/, last access Dec 2015.

[17] Obeo, "Uml designer," Online, 2012, http://www.umldesigner.org/, last access Dec 2015.

[18] A. Danial, "CLOC," Online, 2006, https://github.com/AlDanial/cloc, last access Dec 2015.

[19] K. Kawaguchi, "Jenkins," Online, 2011, https://jenkins-ci.org/, last access Jan 2016.

[20] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Towards automatic scenario generation from coverage information," in *2013 8th International Workshop on Automation of Software Test (AST)*, San Francisco, CA, May 2013, pp. 82–88.