International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-9 Issue-3, February 2020

xUnit-like TT4V: Text-based Testing Environment for Voice-based Services

Hoijin Yoon

Abstract*: Like traditional software products, voice-based services should be tested. However, during testing, the service is activated and executed by a spoken command. Speaking the command for every test case would be very time consuming. TT4V saves testing time by enabling text-form test cases instead of voice-form test cases. It implements an xUnit-style interface. xUnit has been a de facto standard framework for testing tools for many years. Without TT4V, we roughly measured the time to speak 55 commands, listen to 55 responses, and then verify the correctness of the responses. First, we ran five of the 55 test cases without TT4V, and it took 28.03 seconds. For 55 test cases, it would be multiplied by 11 roughly. We ran 55 test cases using TT4V. It took only 0.61 seconds from executing the test cases to showing the analysis of the testing results.

***Keywords*: Voice-based services, unit testing, xUnit framework, test automation**

I. INTRODUCTION

Many voice-based services have become embedded in our daily lives [1], since Google and Amazon released the APIs supporting their voice-based service platforms: Google Assistant and Amazon Alexa. These platforms work through voice-recognition speakers (e.g., Amazon Echo or Google Home), and their services are deployed on their own cloud-server systems, e.g., Amazon's Lambda. Google and Amazon provide some default services on their cloud systems. Moreover, users can develop their own services by importing the APIs provided by Amazon or Google.

Voice-based services should be tested, just like traditional software products. Dynamic testing is done by executing the SUT (Software Under Test) with a test case, which consists of a test input and its expected output [2]. In dynamic testing, the SUT is executed with a test case, and its actual output is asserted to be the same as the expected output defined in the test case.

In the development process, testing is a major and popular quality assurance tool. The more test cases that are applied, the more faults can be detected; however, the greater the testing cost. That is why test designers try to select powerful test cases that can trigger existing faults to be exposed as failures. The test-case selection criteria are key factors that lead to effective and efficient testing. Through the criteria, a set of test cases is selected, and expected to detect as many faults as possible.

Imagine a test case for a voice-based service. The test input would be a voice command, and the expected output would be a spoken response or a specific reaction. For the testing, we *speak* a voice command into a namely AI speaker, e.g., Echo or Google Home, as a test input. Then we *listen* to the speaker to determine whether the response matches the expected output of the test case. This "*Speak-and-Listen*" process takes some time, depending on how fast the human can speak and listen. This decreases the testing performance and is a barrier to adapting the testing process to voice-based services.

To address this *Speak-and-Listen* delay, this study proposes and develops a testing environment where voice-based services can be tested with text-form test cases instead of voice-form test cases. It is named *TT4V*, an abbreviation of text-based testing environment for voice-based services. Moreover TT4V implements *xUnit*-style interfaces for the user's convenience.

Section II describes two major platforms for voice-based services: Google Assistant and Amazon Alexa. In Section III, we introduce the AIY project, a small platform for Google Actions. Section IV explains how our *xUnit*-like *TT4V* works and Section V concludes by discussing TT4V's contributions to testing voice-based services.

II. VOICE-BASED SERVICE PLATFORMS

A. Google Assistant

Google provides a development environment for voice-based services and its APIs, working under a special platform named Google Assistant [3]. Assistant accepts a voice message and decides how the service should react. The services are activated by saying, "Okay, Google." In a scenario where a voice message plays the role of test input data, the response generated by the service should match the expected output of the test case.

B. Amazon Alexa

The Amazon Alexa skill kit implements the *Intent model*, which maps all possible outcomes for a voice invocation [4]. To *publish* a skill means to deploy its intent model on the platform. Once a voice message invokes its corresponding skill through *Echo*, the voice message is transformed to a text form; then, the intent model finds an outcome that matches the text-formatted voice message. The outcome is transformed to voice form and sent back to the Echo speaker.

III. AIY PROJECT

Google's AIY project allows users to build their own natural-language processor and connect it to the Google Assistant or Cloud Speech-to-Text service. This allows the user to ask questions and issue voice commands to their programs. All of this fits in a small cardboard cube, powered by a Raspberry Pi [5].

Revised Manuscript Received on February 05, 2020.
* Correspondence Author
Hoijin Yoon, Department of Computer Engineering, Hyupsung University, Hwaseung, Kyunggi, South Korea. Email: hjyoon@uhs.ac.kr

AIY supports two separate versions: a voice kit and a vision kit. The voice kit is a *Do-it-yourself* intelligent speaker implementing voice recognition and the Google Assistant.

The vision kit is a *Do-it-yourself* intelligent camera implementing image recognition using neural networks. We used the voice kit for testing voice-based services. Fig. 1 shows how the AIY system works with Google Assistant.

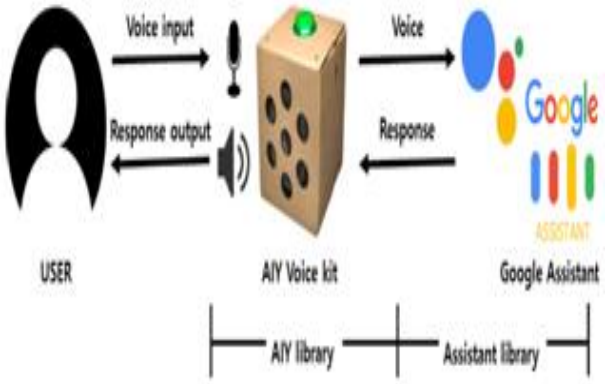

Fig. 1. Google Assistant and AIY

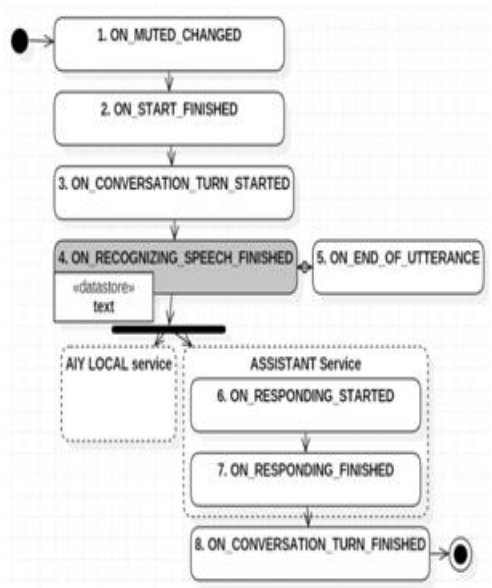

Fig. 2 Event Flow in Google Assistant

The voice input is transformed to text before calling the services in Google Assistant. Fig. 2 shows the event flow, where a voice-based service from the AIY voice kit takes the user's requests, executes the services, and responds with answers [6]. The event named "ON_RECOGNIZING_SPEECH_FINISHED" transforms the voice input into its text-form message. The text message is then sent to the service as input data.

IV. TT4V DESIGN

A. Text-based Testing Environment

Suppose that the user's request is in the form of text. The text-form request message can be adopted directly by the Google Assistant service without the transformation process. In other words, the services can be executed with text-form test input.

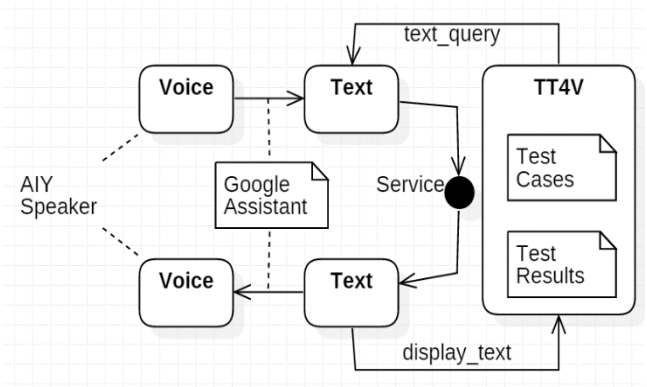

Fig. 3. Voice-based Service Routine and TT4V

The tool, TT4V, includes the code explained above, and its flow diagram is shown in Fig. 3 [7]. TT4V manages test cases and test results. It assigns test input data to the variable, *text_query*. The result of the service execution of the text input is also in the form of text, which is assigned to the variable, *display_text.*

B. xUnit-like Interface

In 1997, Erich Gamma and Kent Beck built a simple and effective framework for unit testing. The framework was intended for testing Java programs, so it was called JUnit [8]. JUnit followed the SUnit framework for Smalltalk [9], previously developed by Erich Gamma. JUnit was released under the Common Public License and hosted on SourceForge. JUnit became a popular framework for developing unit tests in Java [10]. Its successor, known as xUnit, is available for ASP, C++, C#, Eiffel, Delphi, PHP, Python, Smalltalk, and Visual Basic.

xUnit helps developers easily do unit testing and manages test cases as a test suite, as follows. First, it manages test codes as test methods, to distinguish from the source code. Second, it calls assertion methods that include an expected output. Third, it collects the test codes in a test suite to run as a batch. Fourth, it reports how many test codes of the current test suite failed or succeeded [11].

The following definitions are used in xUnit tools [10]:

- *TestCase*—A class that contains one or more tests, named test*XXX* methods. When we mention a *test,* we mean a test*XXX* method; when we mention a *test case*, we mean a class that extends TestCase; i.e., a set of tests.
- *TestSuite (or test suite)*—A group of tests. For example, if a test suite is not defined for a TestCase, xUnit automatically provides one that includes all the tests found in the TestCase.
- *TestRunner (or test runner)*—A test-suite launcher. xUnit provides several test runners that can be used to execute tests.

V. IMPLEMENTATION

A. Voice to Text

Although, in reality, the user would speak the input command to the two-way speaker, TT4V sends the input command in the form of text, as explained in Fig. 3.

To do this, TT4V calls the voice-based service with the text-form command, instead of the speech-based command. The following code illustrates the idea, which uses the input data in text form instead of voice form.

```
display_text = assistant.assist(text_query)
if display_text:
  if check_text.lower() in
display_text.lower():
     All_count = All_count+1
     success_count = success_count+1
     outf.write("%s. success\n" %All_count)
  else:
     All_count = All_count+1
       outf.write("%s. fail\n" %All_count)
```

assistant.assist's argument is *text_query*, which is the input request in the form of text. The *text_query* is sent to Google Assistant, and an appropriate service catches the text query and returns the execution result to the *display_text* variable.

B. xUnit-style TestCase Management

In TT4V, a *TestCase* is a row in a comma-separated values (CSV) file named TestCaseTID.csv. Before launching TT4V, the testers create TestCaseTID.csv and write a test including the TID, test input (input command), and expected output. Each test is written on one row, and a set of rows would be a *TestCase*, in xUnit terms. Moreover, we can group multiple rows as a *TestSuite*, according to the xUnit concept.

The CSV file is selected by clicking a button on the initial window in TT4V. Once the TestCaseTID.csv file is selected, TT4V sends the tests one by one to Google's voice services and receives their response. Then, it compares the responses and the expected output, which are expected to be the same, unless the SUT has a fault in it. TT4V saves the result of this comparison process in a file, output.csv, and shows the result graphically, using the *pytest* library. The library plays the role of *TestRunner*. The following code shows how the *pytest* library works in TT4V.

```
tests=[]
f=open("/home/tt4v/Downloads/(Source Code)
TT4V/B-Model/output.csv",'r',encoding='utf-8'
)
rdr= csv.reader(f)

for line in rdr:
   tests.append(line)

 f.close()

 cases=[]
 for i in range(len(tests)):
     cases.append(tests[i][4])

 @pytest.mark.parametrize('result',cases)
 def test_Compare(result):
   assert result == 'True',"Fail!"
```

C. GUI with an Example

TT4V starts with empty panels in a window, as shown in Fig. 4. Note that we have already created a .CSV file with the test cases. In this example, the file name is TestCaseTID.csv.

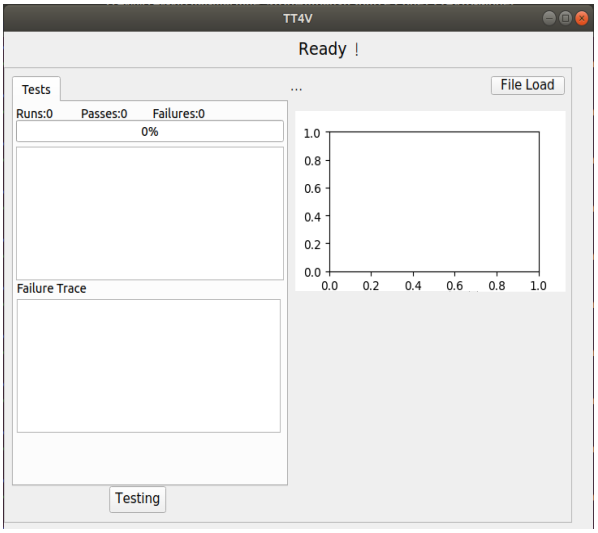

Fig. 4 TT4V's Initial GUI

We start a test by clicking the "File Load" button, shown in Fig. 4, which opens the CSV file with the previously written test cases. A test case includes the input command and the expected output. The input command is supposed to be spoken by a human to call voice-based services; however, TT4V calls the voice-based services with the written input command, without requiring a human to speak. Once the "Testing" button on the bottom of Fig. 4 is clicked, TT4V executes the SUT with the test cases.

For example, we wrote 55 test cases and tested them using TT4V. It sent test data to the matching Google services, received their responses, and compared them with the expected output. The testing result would be marked "fail" if the comparison result was "not-equal." It would be "pass" if the comparison was "equal." TT4V analyzes these pass-or-fail results, and presents them in *xUnit* style, as shown in Fig. 5.

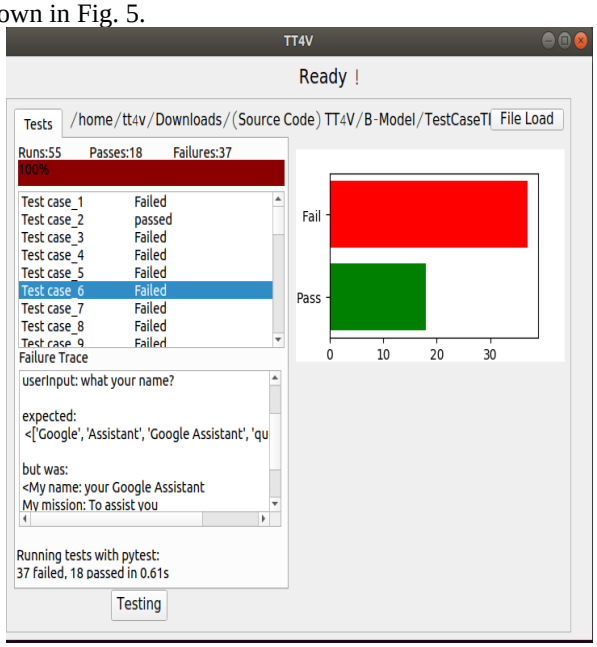

Fig. 5. UI After Testing

TT4V ran 55 test cases, as shown in Fig. 5. Eighteen of them passed; the others failed. Like JUnit, TT4V shows the details of a selected test case in the bottom pane.

VI. RESULT ANALYSIS

We ran 55 test cases using TT4V, and it took only 0.61 seconds from executing the test cases, as shown in the bottom of Fig. 5. It looks very quick for testing 55 cases. For making a detail comment, we roughly measured the time to speak input commands, to listen to their responses, and then to decide if the responses are the same as expected. We did it on only 5 test cases without TT4V, and we measured the time to take those three steps, which are to speak, to listen, and to decide. The time was 28.03 seconds. It was for only 5 test cases, therefore, it would be multiplied by 11 roughly for 55 test cases. It was 308.55 seconds. TT4V saves 307.94 seconds approximately.

Table I. Comparison of "with-TT4V" and "without-TT4V"

	with TT4V	without TT4V
# of Test cases	55	55
Time to Run	0.61 seconds	308.55 seconds
Preprocessing (Writing TC file)	Required	Not-required
Testing Repetition	Possible	Not-possible
Human mistake	Not-acceptable	Can-happen

As described in Table I, TT4V requires a test cases file that should be written before testing. It would be a burden of using TT4V at first, but the written test case file enables the testing repetition. Testing can be always re-done with the test cases with the csv file. Moreover, mistakes made by Human cannot happen in TT4V since the tool automatically covers all the testing activities*; Speak, Listen, Decide*.

VII. CONCLUSION

Like any software product, voice-based services need to be tested. However, speaking the command for every test case is very time consuming. Hence, we developed TT4V. It enables voice-based services to be tested with text-form test cases.

TT4V is an xUnit-style tool. xUnit provides methods for building tests and test suites, and manages the test codes separately from the SUT source code. It analyzes the testing results, showing a ratio of fails or passes and the details of each test case.

Unlike xUnit, TT4V automatically generates the test codes. It uses the set of test cases in a CSV file written by the user. We utilized the *pytest* library for this automatic generation.

As shown at the bottom of Fig. 5, it takes only 0.61 seconds to test 55 test cases. Imagine the time it would take to speak 55 commands, listen to 55 responses, and then verify the correctness of the responses. TT4V contributes to saving testing time by enabling text-form test cases, instead of voice-form test cases.

ACKNOWLEDGMENTS

This work was supported by the Hyupsung University Research Grant of 2019.

This research was also supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2017R1D1A1B03034557).

REFERENCES

1. Cathy Pearl, *Designing Voice User Interfaces*, O'Reilly, 2017.
2. Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2016.
3. Google Assistant, https://assistant.google.com/.
4. Amazon Alexa, https://www.amazon.com › alexa-skills.
5. AIY page, https://aiyprojects.withgoogle.com.
6. Wonuk Cha and Hoijin Yoon, "An Analysis on Events-Flow of Google Assistant in AIY project," in *Proceedings of the Korea Computer Conference*, Jun. 2018.
7. Eungjun Kim, Hoijin Yoon, and Wonuk Cha, "Text-based Testing Environment for Voice-based Services," in *Proceedings of APSEC*, Dec. 2018.
8. Kent Beck and Erich Gamma, "Test-infected: Programmers love writing tests," *More Java Gems*, 2000.
9. Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.
10. Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory, *JUnit in Action*, Manning, 2010.
11. Gerard Meszaros, *xUnit Test Patterns*, Pearson Education, 2007.

AUTHOR'S PROFILE

Hoijin Yoon is currently an associate professor in the Department of Computer Science and Engineering at Hyupsung University since 2007. She received a B.S. degree in Computer Science from Ewha Woman's University in Korea, and M.S. and Ph.D. degrees in Computer Science and Engineering from Ewha in 2004. Her research interests are in software engineering with particular emphasis on testing. Her current research project is on the verification and validation of interactions between intelligent things and humans.