REPRODUCING FIELD FAILURES BASED ON SEMI-FORMAL FAILURE SCENARIO DESCRIPTIONS

A Thesis

by

Gün Karagöz

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Electrical and Electronics Engineering

Özyeğin University
June 2015

REPRODUCING FIELD FAILURES BASED ON SEMI-FORMAL FAILURE SCENARIO DESCRIPTIONS

Approved by:

Assist. Prof. Dr. Hasan Sözer
Department of Computer Science
Özyeğin University

Assoc. Prof. Dr. Çağatay Çatal
Department of Computer Engineering
İstanbul Kültür University

Assist. Prof. Dr. Barış Aktemur
Department of Computer Science
Özyeğin University

Date Approved: 27 May 2015

To my dear parents Ziynet and Onur Karagöz for their material aid and spiritual support, and to my dear wife Şerife for her endless love, support and encouragement...

ABSTRACT

Due to the increasing size and complexity of software systems, it becomes hard to test these systems exhaustively. As a result, some faults can be left undetected. Undetected faults can lead to failures in deployed systems. Such failures are usually reported from the field back to developers. It requires considerable time and effort to analyze and reproduce the reported failures because their descriptions are not always complete, structured and formal. In this study, a novel approach for automatically reproducing field failures to aid their debugging is introduced. The approach relies on semi-structured failure scenario descriptions that employ a set of keywords. These descriptions are pre-processed and mapped to a set of predefined test case templates with valid input sets. Then, test cases are generated and executed to reproduce the reported failure scenarios. The approach is evaluated with an industrial case study performed in a company from telecommunications domain. Many field failures were successfully reproduced. The approach is also adopted in the quality assurance process of the company. After one-time preparation of reusable test case templates and training of test engineers, 40% of the reported failures were reproduced without any manual effort.

ÖZETÇE

Yazılım sistemlerinin artan hacmi ve karmaşıklığı nedeniyle bu sistemleri her ayrıntısına kadar test etmek zorlaşmaktadır. Sonuç olarak, bazı hatalar tespit edilemeden kalabilir. Tespit edilmemiş hatalar canlı sistemlerde güvenilirliğe tehdit oluşturabilir. Bu hatalar genellikle sahadan yazılım geliştiricilere raporlanır. Raporlamalar her zaman tam, açık ve biçimsel olmadığı için hatalı kullanım senaryolarını analiz edip yeniden üretmek hatrı sayılır bir zaman ve emek gerektirir. Bu çalışmada, hata ayıklamaya yardımcı olmak amacıyla sahadan bildirilen hatalı senaryoları otomatik olarak yeniden meydana getirmek için özgün bir yaklaşım sunulmaktadır. Yaklaşımımız, anahtar kelime kümesi içeren yarı-yapısal hata senaryolarına dayanmaktadır. Bu açıklamalar otomatik olarak çözümlenerek, geçerli girdi kümeleriyle önceden tanımlanmış test senaryosu şablonlarına eşlenir. Sonrasında, raporlanan hata senaryolarını yeniden üretmek için test senaryoları üretilir ve işletilir. Bu yaklaşım telekomünikasyon sektöründen bir şirkette yapılan bir sanayi vaka çalışması ile değerlendirildi. Sahadan raporlanan birçok hatalı senaryo başarıyla yeniden üretildi. Yaklaşım aynı zamanda şirketin kalite güvence sürecinde benimsenmiştir. Tekrar kullanılabilir test senaryosu şablonlarının bir kerelik hazırlanmasının ve test mühendislerinin eğitimlerinin ardından, raporlanan hataların %40'ı manüel çaba gerektirmeksizin yeniden üretilmiştir.

ACKNOWLEDGMENTS

This work is partially supported by P.I.Works. We would like to thank software developers and software test engineers at P.I.Works for sharing their code base with us and supporting our case study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The increasing size and complexity of software systems make it hard to prevent or remove all possible faults. It is not feasible to test these systems exhaustively due to lack of resources. As a result, they may be delivered and deployed together with a set of undetected faults. These faults lead to errors and eventually to failures that are exposed to the users of the system [1]. After observing a failure, users usually report an issue that describes the observed failure scenario. In practice, many of such issues are collected from the field. They are analyzed by software developers to reproduce the corresponding failures and find the root causes, i.e., faults. This debugging process often requires significant time and effort because reports are usually not complete, structured and formal.

There exist capture-replay tools [2, 3, 4, 5, 6] that can be used for replaying execution scenarios as observed by users. However, these tools are hardly utilized in practice due to their performance overhead and their burden on the users for controlling the tool. One can automatically generate test cases if the issue reports are formal and complete. However, users of the system usually lack the necessary skills and/or they are reluctant to use a formal notation as concluded in a recent survey [7].

In this thesis, we propose an approach that relies on semi-formal scenario descriptions of field failures to aid their debugging. These descriptions follow a structure and employ a set of predefined keywords. We developed a tool set that can analyze such descriptions and pre-process them to remove minor mistakes such as typos. After the pre-processing step, they are transformed into formal descriptions. These descriptions are mapped to test case templates. These templates are predefined in the form of a library. They are defined and organized according

to the system use cases and they include valid test inputs necessary to execute a scenario. If a description can be mapped to a template, concrete test cases are generated and executed to reproduce the reported failure scenarios.

The approach is evaluated in the context of an industrial case study from telecommunications domain. Users of a real system were instructed on the use of semi-formal scenario descriptions to report issues. Then, we collected all the reports for the system, including the previously created ones that do not follow any structure at all. 60% of the reports could not be utilized due to lack of adherence to the expected structure or because, they could not be mapped to a test case template. However, a total of 160 reports were successfully processed and mapped. As a result, 160 test cases were automatically generated and executed. As such, the corresponding failures were successfully reproduced without any manual effort.

The main contributions of this study are twofold. First, we introduce an approach for automatically generating test cases based on semi-formal failure scenario descriptions collected from the field to reproduce failure scenarios as observed and reported by users. Second, we evaluate the approach and present promising results based on an industrial case study. Our approach can significantly reduce the time and effort required for analyzing reports and debugging reported failures. In addition, a set of reusable test cases are automatically created without manual effort. This is a complementary approach to manual test case creation based on functional requirement specifications and model-based test case generation [8, 9].

The remainder of this paper is organized as follows. In the following chapter, we introduce tools, techniques and languages that are employed in our approach. In Chapter 3 we summarize the related studies. In Chapter 4, we present the problem context and an industrial case study from the telecommunications domain. This case study is used as a running example, while we explain our approach in Chapter 5. We present the results of the case study in Chapter 6. We discuss the results and limitations of the approach in Chapter 7. Finally, in Chapter 8, we conclude the paper and discuss possible extensions of our work.

CHAPTER II

BACKGROUND

In this chapter, we introduce the background knowledge regarding a set of tools, techniques and languages, which are employed in our approach.

There is an inherent communication gap between developers and users as well as Quality Assurance (QA) teams and business stakeholders. This gap can be shortened by means of domain-specific languages (DSLs); however, some users can be reluctant to describe the expected or error-prone system behavior with a DSL. Therefore, we employ a simpler structure for issue descriptions. This structure is based on the *Gherkin* format [10]. We introduce a pre-processing tool that can parse issues and convert them into *Gherkin* format, while handling mistakes such as typos.

Gherkin is a scripting language for specifying system usage behavior. Hereby, each feature of the system is defined by means of examples, namely scenarios, and every scenario comprises a list of steps, which must start with one of the keywords *Given, When, Then, But* or *And*. As such, it employs a keyword-driven approach [11]. A sample feature description is provided in Listing 2.1.

Gherkin format has been mainly employed to define test cases in Behavior-driven development (BDD), which evolved from Test-driven development (TDD). It is originally used for describing usage scenarios to specify features in a way that stakeholder communication and test automation is increased [12]. In our approach,

Listing 2.1: A sample issue description in Gherkin format

```
1 Feature: Exporting Report
2   Scenario: Exporting in PDF format
3     Given I run report 'MyReport'
4     When I export in PDF format
5     Then Report in PDF format should be downloaded
```

we utilize Gherkin format to describe failure scenarios in addition to feature descriptions. This enables automated generation and execution of test cases that reproduce failures. To derive executable test cases from scenario descriptions, every step of the scenario should be mapped to an executable test step.

Listing 2.2: Sample step definitions based on the scenario steps involved in the issue description presented in Listing 2.1

```
1  using System;
2  using TechTalk.SpecFlow;
3
4  namespace PIWeb_UAT
5  {
6      [Binding]
7      public class ExportingReportSteps
8      {
9          [Given(@"I run report '(.*)'")]
10         public void GivenIRunReport(string p0)
11         {
12             ScenarioContext.Current.Pending();
13         }
14
15         [When(@"I export in PDF format")]
16         public void WhenIExportInPDFFormat()
17         {
18             ScenarioContext.Current.Pending();
19         }
20
21         [Then(@"Report in PDF format should be downloaded")]
22         public void ThenReportInPDFFormatShouldBeDownloaded()
23         {
24             ScenarioContext.Current.Pending();
25         }
26     }
27 }
```

We utilized the *SpecFlow* framework[1] for parsing descriptions in Gherkin format and creating test steps associated with the scenario steps. Listing 2.2 shows a set of sample step definitions that are generated by SpecFlow based on the description presented in Listing 2.1. For instance, the method named *GivenIRunReport* at Line 10 in Listing 2.2 is associated with the scenario step defined at Line 3 in Listing 2.1. Hereby, the name of the report is defined as an argument (parameter) for the method.

[1] http://www.specflow.org/

Table 1: Sample Selenium commands and descriptions.

Definition	**Command**	**Argument(s)**
Selenium control command for wait operation	setTimeout	duration (miliseconds)
Browser operation for opening a link	open	url
Browser operation for locating a UI element	findElement	locator[3]
Keyboard operation for typing a text	type	locator, text
Mouse operation for clicking on a UI element	click	locator
Mouse operation for checking a radio button or checkbox	check	locator
Verification operation for checking element availability	verifyElementPresent	locator

SpecFlow is an open source framework. It enables binding of usage scenarios to executable test code [13]. However, the generated test code is rather a template, which have to be instantiated with concrete test steps. For instance, we can see three methods defined in Listing 2.2. The contents of these methods (at lines 12, 18 and 24) are yet to be defined. In our work, we focus on Web-based applications and we have utilized the *Selenium* tool[2] to define the necessary concrete test steps.

Selenium is a software testing framework for web browser automation. It allows to record and replay browser actions to simulate user behavior on the graphical user interface (GUI). It also provides a library that includes methods associated with possible user actions through the GUI. Sample Selenium commands and descriptions are shown in Table 1.

[2]http://www.seleniumhq.org

[3]Selenium keyword to find and match the UI elements on page to interact with. UI elements can be *located* using Id, Name, Link, DOM, XPath, or CSS.

We used a Mozilla Firefox Plugin of the Selenium tool to capture GUI elements for collecting possible user actions such as clicking on a button and typing in a field. Then, step definitions are populated with the corresponding method calls provided by Selenium. Listing 2.3 shows the resulting step definitions in terms of method calls, whereas Listing 2.4 shows the implementation of these methods by using the Selenium API.

Listing 2.3: Step definitions implemented as method calls

```
using System;
using TechTalk.SpecFlow;

namespace PIWeb_UAT
{
    [Binding]
    public class ExportingReportSteps
    {
        [Given(@"I run report '(.*)'")]
        public void GivenIRunReport(string p0)
        {
            selenium.runReport(p0);
        }

        [When(@"I export in PDF format")]
        public void WhenIExportInPDFFormat()
        {
            selenium.exportReportInPdf();
        }

        [Then(@"Report in PDF format should be downloaded")]
        public void ThenReportInPDFFormatShouldBeDownloaded()
        {
            selenium.assertPdfFileDownloaded();
        }
    }
}
```

Our approach relies on a chain of tools to reproduce field failures. These tools include a parser to process issue reports, and two external tools, SpecFlow and Selenium, to generate and execute test cases, respectively. The usage of these tools will be explained in more detail as part of the approach description. First, in the following chapter, we introduce a case study that will be used as a running example.

Listing 2.4: Method implementations for scenario steps using the Selenium API

```
1  public class selenium
2  {
     ...

3    public IWebElement TextBox_searchReport()
4    {
5      return driver.findElement(By.Id("SearchReport"));
6    }

8    public IWebElement Button_runReport()
9    {
10     return driver.findElement(By.Id("RunButton"));
11   }

     ...

13   public void runReport(string reportName)
14   {
15     seleniumwd.open("http://localhost/PIWeb");
16     seleniumwd.waitForElement(TextBox_searchReport);
17     seleniumwd.type(TextBox_searchReport,reportName);
18     seleniumwd.click(Link_SearchResult);
19     seleniumwd.waitForElement(Button_runReport);
20     seleniumwd.click(Button_runReport);
21   }

23   public void exportReportInPdf()
24   {
25     seleniumwd.click(Button_exportInReport);
26     seleniumwd.check(Check_pdfRadio);
27     seleniumwd.click(Button_exportInExportWindow);
28     seleniumwd.click(Button_downloadExport);
29   }

31   public void assertPdfFileDownloaded()
32   {
33     report(isFileDownloaded(downloadPath, fileName));
34   }

36 }
```

CHAPTER III

RELATED WORK

Previously, the use of NLP techniques were discussed [14] for deriving formal descriptions from informal issue reports. In that study, it is assumed that the reports are defined purely with a natural language, without following any structure. Advanced NLP techniques are applied to obtain formal descriptions based on such specifications. We observed that this approach is not practical for our case. It was not feasible to obtain formal specifications from completely unstructured, informal specifications. Therefore, we relied on semi-formal descriptions instead.

Keyword-driven test automation systems are studied [15] to improve efficiency and re-usability of test case management. There also exist open source tools like Fitnesse[1] and the Robot Framework[2] that support keyword-driven testing methodology. We also employed a keyword-driven technique by taking advantage of the parametrized input capability of the Gherkin Language. Nevertheless, we have populated our test inputs based on user inputs and model test cases based on failure scenarios, contrary to the modeling approach adopted in other keyword-driven approaches [16] [17].

Automated generation of test cases from UML models was proposed in [9]. UML diagrams that represent use cases together with an object constraint language dictionary are employed to create test cases. In our case, such an approach is not applicable since issue reports are provided in a textual format. System users do not have knowledge on the required formalism and graphical notations.

A promising tool to reproduce fields failures for debugging was presented [18]. This tool performs analysis at the source code level to reproduce field failures

[1]http://www.fitnesse.org
[2]http://robotframework.org

by using complementary information such as memory dumps. Moreover, continuing research [19] extends the work by genetic programming-based solution for programs with structured inputs. In these works, we address the same problem; however, we rely only on the available issue reports and we deal with addressing problems on user level.

There exist test execution automation tools [20] that employ BDD and a set of open source tools such as SpecFlow. However, test case generation is not in the scope of these tools. In this work, we focus specifically on test case generation to reproduce failures as described in issue reports.

To the best of our knowledge, the use of semi-formal issue reports to reproduce field failures has not been investigated before. In this work, we integrated this approach to the issue management process of a company and obtained promising results based on real issue reports received from system users.

CHAPTER IV

PROBLEM STATEMENT

In this chapter, we present the problem context and an industrial case study that will be used as a running example for both motivating the problem and illustrating the approach later on.

Our case study is a Web-based system called P.I.Web that is developed and being maintained by P.I.Works Inc.[1]. P.I.Works is a company in telecommunications industry which offers mobile network performance monitoring and optimization solutions for telecommunications operators. We have selected P.I.Web as our case study among three products of the company since there were reported issues available for this system and the system was still under maintenance.

P.I.Web is a web-based product that allows users to create and execute reports using a set of so-called Key Performance Indicators (KPI) in order to monitor network performance. A sample report screen is shown in Figure 1. In this screen, the header section contains menu items for creating, managing, searching reports as well as user settings, help and log-out action buttons. The left sidebar is used for network element selection to define the report scope. Report parameters such as date range, KPI selection and report action buttons such as run, filter, edit threshold, export, are displayed on top of the central frame.

[1] http://www.piworks.net

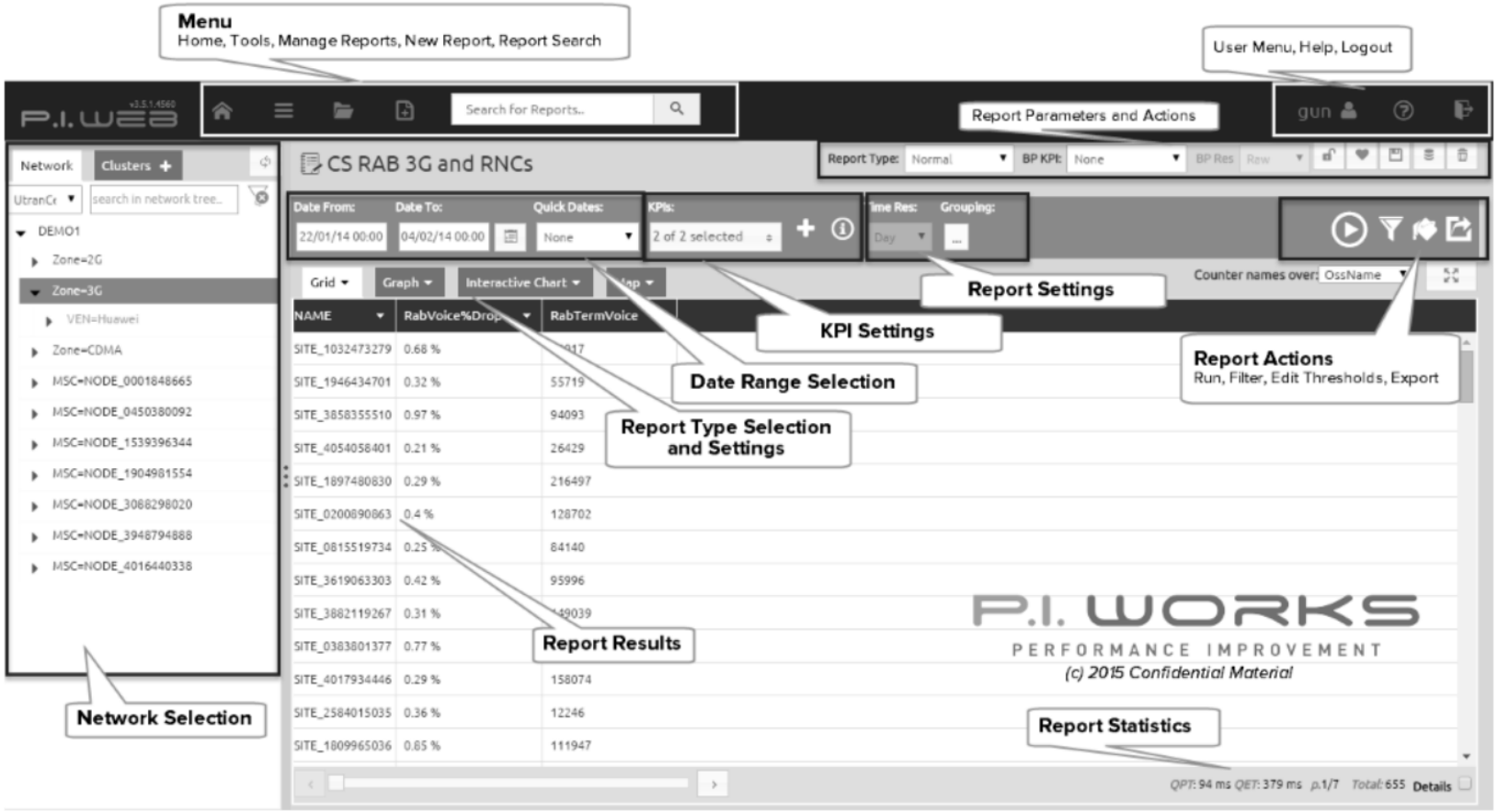

Figure 1: Report user interface of the P.I.Web system.

Fundamental use cases of PIWeb are depicted with a UML Use Case Model in Figure 2. Users can generate performance reports with P.I.Web based on a lot of options such as network element selection, date range, report resolution granularity, grouping, KPI selection etc. In addition, users can extend the list of KPIs with custom formulated KPIs. This leads to an open-ended, infinitely many variations in option selections that may be used in report generation. Besides, users can engage with a set of use cases in sequence. All options and their permutations can result in millions of possibilities since repeating actions in different orders may also lead to different results.

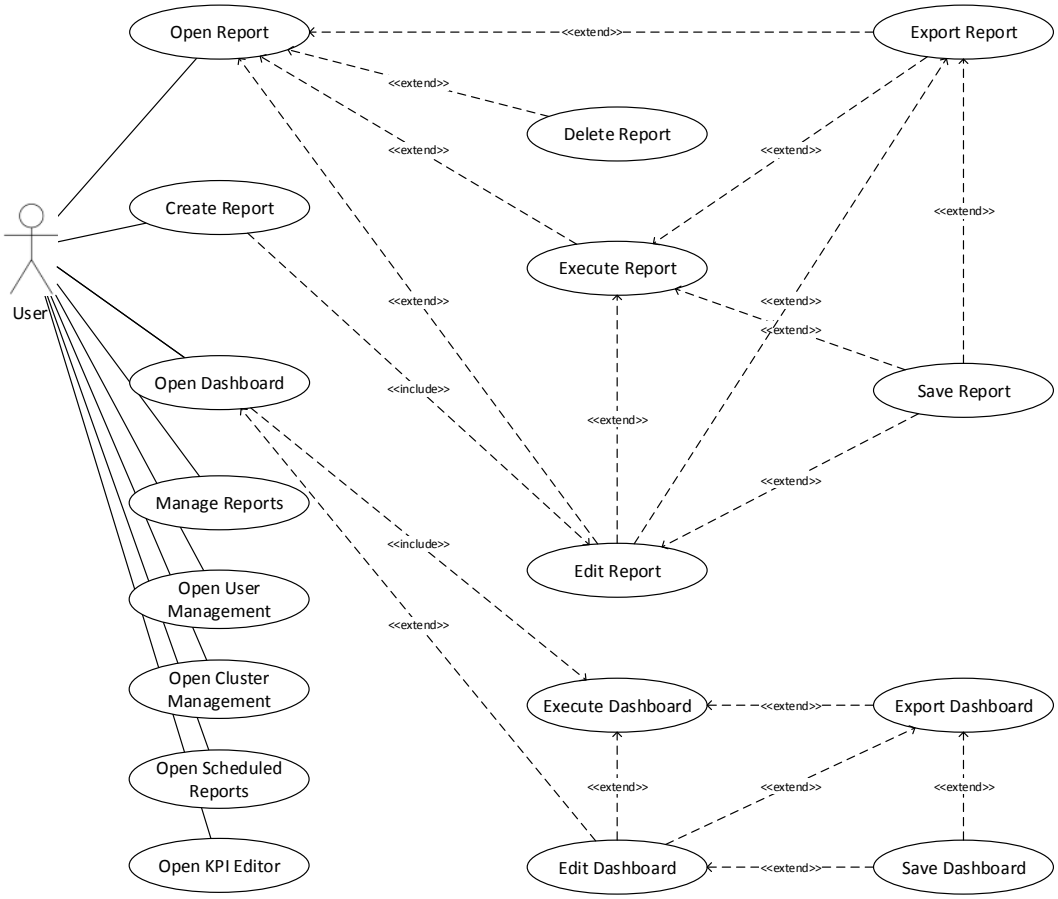

Figure 2: UML use case model for the P.I.Web system.

Therefore, it is not feasible to attain full coverage in functional testing by just considering usage scenarios according to specifications. Moreover, other customer-specific options can make it even more challenging to test the whole system. Hence, it is inevitable to miss some of the faults, which are reported by system users later on.

P.I.Works is using a ticketing system to track the reported failure reports. These reports are verified and analyzed by the QA team to reproduce the reported failures and add new test cases to the test suite that captures these failures.

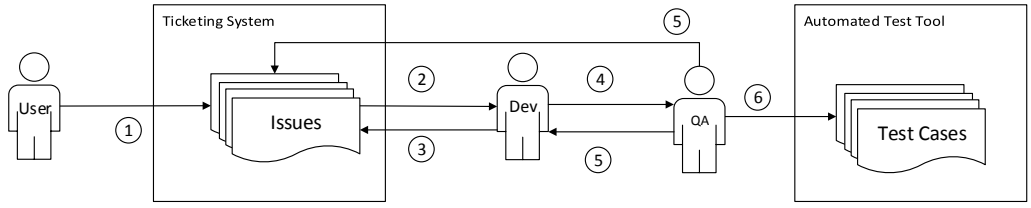

Figure 3: Current failure reporting/fixing cycle.

Figure 3 shows the current traditional cycle of a failure reporting and fixing process. Both software test engineers and end users of the system[2] provide failure reports, i.e., *issues*, when they are exposed to failures. User reports the observed failure, the followed steps together with additional information such as the software version, build number, module name, database information, module specific parameters. In some cases, user attaches screenshots or video captures to provide more information (1). Issues are assigned to a developer to analyze and fix them. After that, the developer needs to debug the problem to find the root cause of the failure (2). To do so, the developer needs to open, read and analyze the assigned issue in detail, check attachments, and *reproduce* the issue in the development environment (3). After fixing the issue, the developer assigns the issue to the QA team to verify the incorporated fix (4). QA team tries to *reproduce* the reported issue with the latest build of the software system. If it is confirmed that the issue is fixed, the issue is closed. Otherwise, the issue is reopened and assigned back

[2]We refer to both as *users*.

to the developer (5). If the fix is verified and the issue is closed, the QA team enriches the test suite for the system by adding a new test case corresponding to the fixed issue (6). The execution of this test suite is automated with an in-house developed test automation tool.

Failure scenarios that are described in issues need to be carefully analyzed to be able to reproduce failures, find the corresponding faults and fix them. Issue reports need to be defined clearly, so that developers can understand and reproduce the problem. Well-defined issue descriptions comprise all of the steps involved in the error-prone usage scenario and the relevant configuration information. However, issue descriptions provided by users in practice are mainly informal descriptions. Thus, developers spend considerable time just to understand reported issues. This manual and time-consuming debugging process requires significant effort and this effort often exceeds the effort spent for fixing the problem. In the following section, we introduce our approach to improve this process.

CHAPTER V

APPROACH

In this chapter, we bring in our approach in details. Figure 4 shows our proposed approach. Users report issues using the ticketing system (1). Issues describe the failure scenario and the system settings during the failure. Issues are grabbed by *Automated Test Case Generator* and assigned to developers for analysis and bug fix stage (2). *BDD Parser* extracts required information from the issue to create a failure scenario description in Gherkin format while handling possible typos. *Test case generator* creates a new test case using parameters and sequence of scenario steps in the description by driving the corresponding methods in the library of *Scenario Templates*. This library contains step definitions and executable test steps for each user action (3). The generated test case is provided to the automated test execution tool as part of the extended test suite.

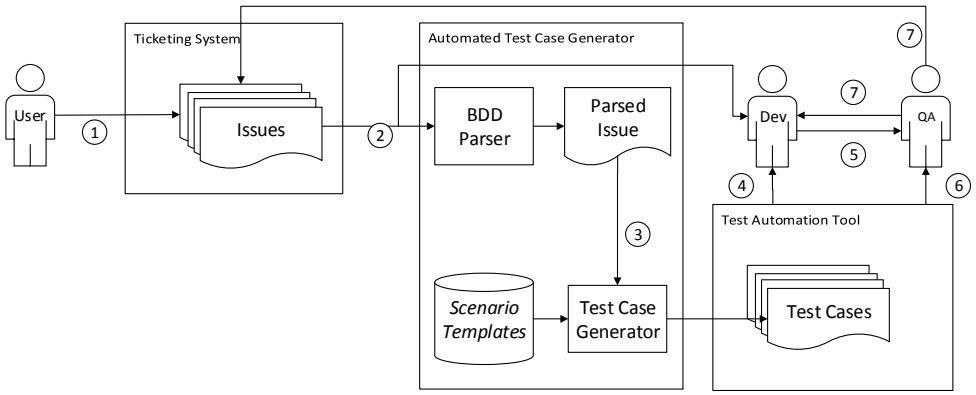

Figure 4: The overall approach for failure reporting/fixing cycle.

Responsible developer analyzes the issue and reproduces it via *Automated Test Tool* in latest build of the software (4). The developer fixes the issue after finding the root cause. The developer assigns the issue to the QA team for verification of the fix (5). QA team reproduces the issue using *Test Automation Tool* on the build

released after the fix (6). QA process continues (7) according to the traditional cycle as described in the previous chapter except manual update of test case list. Since new test cases are added automatically by the tool, QA team is not required to handle this process manually.

In the following sections, we explain each step of our approach in detail. We also provide example artifacts from our case study. We can not provide these artifacts completely due to confidentiality; however, we present representative examples.

5.1 Preparing Scenario Templates

Scenario templates define possible user interactions via the GUI. Their creation requires a one-time effort and they can be reused for generating multiple test cases. Maintenance is not required as long as the functionality of the system and the corresponding GUI components are not subject to major changes. Scenario templates are organized according to the system use cases as presented in Figure 2.

To create a scenario template, we first define possible user actions for each use case in Gherkin format. A sample use case description is shown in Listing 5.1. Then, we use *SpecFlow* to generate test step definitions for each distinct action in the form of separate methods. Finally, we populate these methods with relevant method calls to Selenium Application Programming Interface (API) such that each user action can be replayed with a concrete method. For instance, the created scenario template corresponding to the use case definition in Listing 5.1 is shown in Listing 5.2. Each method in this scenario template is a test step that corresponds to a user action. The use of arguments/parameters and the arbitrary ordering of these test steps make it possible to reuse the same template to instantiate many different test cases.

When the template is completed, we can map different user actions to concrete methods that replay the action via Selenium. For instance, Line 4, 16 and 28 in

Listing 5.1: A sample use case description for the Report Creation use case

```
Feature: Report Creation Template

Scenario: Report should be created successfully
Given Link 'http://localhost/PIWEB/login.aspx'
And Select KPI Category '2G_CELL'
And Add KPI 'CallDrop%'
And Click Apply
And Network Element 'BSC03'
And Dates '19/01/14 00:00', '22/01/14 00:00'
And Grouping 'Datetime'
When I click execute
Then Report should be executed succesfully
And 'CallDrop%' should be displayed in grid header
And CloseBrowser
...

Scenario: Counter report should display counter alias
Given Link 'http://localhost/PIWEB/login.aspx'

...


And Add Counter 'CounterName'

...


When I click 'Execute'
Then Report grid should display 'CounterAlias'
And CloseBrowser

...


Scenario: Report should be saved as copy
Given Link 'http://localhost/PIWEB/login.aspx'

...


And open report 'Report_Name'

...


When I click 'Save As'
Then Report title should be 'Report_Name Copy'
And CloseBrowser
```

Listing 5.2: Sample template for Report Creation

```
using System;
using TechTalk.SpecFlow;

namespace PIWeb_AutoTest
{

 [Binding]
 public class ReportCreationTemplate
 {

  [Given(@"Link '(.*)'")]
  public void GivenLink(string p0)
  {
   selenium.setUrl(p0);
  }

  ...


  [Given(@"Network Element '(.*)'")]
  public void GivenNetworkElement(string p0,string p1)
  {
        selenium.setNetworkElement(p0,p1);
  }

  [Given(@"Dates '(.*)', '(.*)'")]
  public void GivenDates(string p0,string p1)
  {
        selenium.setDateTime(p0,p1);
  }

  ...


  [When(@"I click '(.*)'")]
  public void WhenIClick(string p0)
  {
   selenium.clickButton(p0);
  }

  [Then(@"Report should be executed successfully")]
  public void ThenReportShouldBeExecutedSuccessfully()
  {
        selenium.assertGridExists();
  }

  ...


  [Then(@"CloseBrowser(.*)")]
  public void ThenCloseBrowser(int p0)
  {
        selenium.closeBrowser();
        selenium.printReport();
  }

 }

}
```

Listing 5.1 are mapped to the same test step (i.e., method at Line 12 in Listing 5.2), which takes an argument as input. Similarly, the *Dates* action at Line 7 in Listing 5.4 is mapped to Line 24 in Listing 5.2 to the *GivenDates* method, which replays the corresponding GUI event.

In principle, there can be more than one scenario template defined per use case. In our case study, we defined 21 templates for 3 different use cases. In the following, we explain the utilization of these templates for generating test cases from issue reports.

5.2 Parsing Ticketing System Issues

Issues in the ticketing system can be parsed and reproduced if and only if there exist a mapping from each failure scenario step to a test step in the defined scenario templates.

Each line of the issue descriptions is parsed by keywords. Different issue descriptions may be mapped to the same Gherkin format definition as displayed in Table 2 to ensure the utilization of same test steps for different strings that point on the same action. For instance, we simply look for *Network Element* or *Network object* keyword and the following argument in a given phrase in order to map this phrase to *Given Network Element* definition.

Table 2: A list of sample Gherkin format conversions for the given issue descriptions.

Description	**Conversion**
Select Network Element BSC03	Given Network Element 'BSC03'
I select network element BSC03	Given Network Element 'BSC03'
I Selected Network element 'BSC03'	Given Network Element 'BSC03'
I choose Network object 'BSC03'	Given Network Element 'BSC03'
...	...
Start Date 19/01/14 00:00 and enddate 22/01/14 00:00	Given Dates '19/01/14 00:00', '22/01/14 00:00'
Set Start date to 19/01/14 00:00 end date to 22/01/14 00:00	Given Dates '19/01/14 00:00', '22/01/14 00:00'
I set date range as 19/01/14 00:00 - 22/01/14 00:00	Given Dates '19/01/14 00:00', '22/01/14 00:00'

Listing 5.3: Sample Reported Issue

```
1 Open Link http://localhost/PIWEB/login.aspx
2 Create a report with category 2G_CELL
3 Add KPI CallDrop%
4 Select Network Element BSC03
5 Select start date 19/01/14 00:00 end date 22/01/14 00:00
6 Select Grouping Datetime, Node
7 Click execute
8 Expected: Report should be executed successfully.
9 Actual: Exception error is displayed : WebServiceError
```

A sample reported issue is provided in Listing 5.3. This issue is related to the same use case as the scenario template introduced in Listing 5.2. For instance, in Line 25 in Listing 5.2 the test step (i.e., method) *GivenDates* which corresponds to the user action for setting date range (Line 5) in Listing 5.3. This method includes calls to relevant methods of the Selenium API. We can also see that it takes two arguments. These arguments are taken from the issue description when instantiating a test case.

BDD parser gets such issues created by system users as input. It parses this input using basic regular expression (RegEx) search techniques [21] to create a formal BDD scenario for handling typos and converting them to parametrized methods such as mapping Line 2 in Listing 5.3 by Line 3 in Listing 5.4. Hereby, RegEx techniques are applied for detecting keywords that are possibly subject to typos, and to obtain scenario steps and relevant parameters in issue descriptions. As such, *BDD parser* can process issue descriptions in natural language format and convert them to a format in Gherkin style. A sample output of the module is presented in Listing 5.4. This output corresponds to the description in Listing 5.3.

5.3 Generating Test Cases from Parsed Issues

All issues that can be parsed by *BDD parser* are added as test cases to the test suite. A sample generated test case is shown in Listing 5.4. A test case is based on the sample reported issue (Listing 5.3) driven by the scenario template in Listing 5.2. In the following, we describe the execution of the generated test cases.

Listing 5.4: Sample BDD Parser Output

```
Scenario: 2G_CELL Report should be created successfully
Given Link 'http://localhost/PIWEB/login.aspx'
And Select KPI Category '2G_CELL'
And Add KPI 'CallDrop%'
And Click Apply
And Network Element 'BSC03'
And Dates '19/01/14 00:00', '22/01/14 00:00'
And Grouping 'Datetime'
And Grouping 'Node'
When I click 'Execute'
Then Report should be executed successfully
And CloseBrowser
```

5.4 Executing Test Cases

P.I.Works uses an in-house developed Test Automation Tool (PIATT) for executing test cases. This tool employs Selenium. The tool operates available test scenarios on a selected browser and test environment, while recording the duration for the completion of every action for collecting performance statistics. PIATT replays GUI scenarios and automatically reports the results with the involved steps together with a screenshot of the GUI for each failing test.

The set of generated test cases is automatically transferred to PIATT via a module in the tool, as shown in Figure 5. Figure 6 shows an example output of the tool for a single test case execution. There is also a recorded demonstration of the overall approach and the tool set available online[1]. In the following section, we discuss the overall results of our case study.

[1]http://srl.ozyegin.edu.tr/projects/piatt

Figure 5: Executing test cases with PIATT.

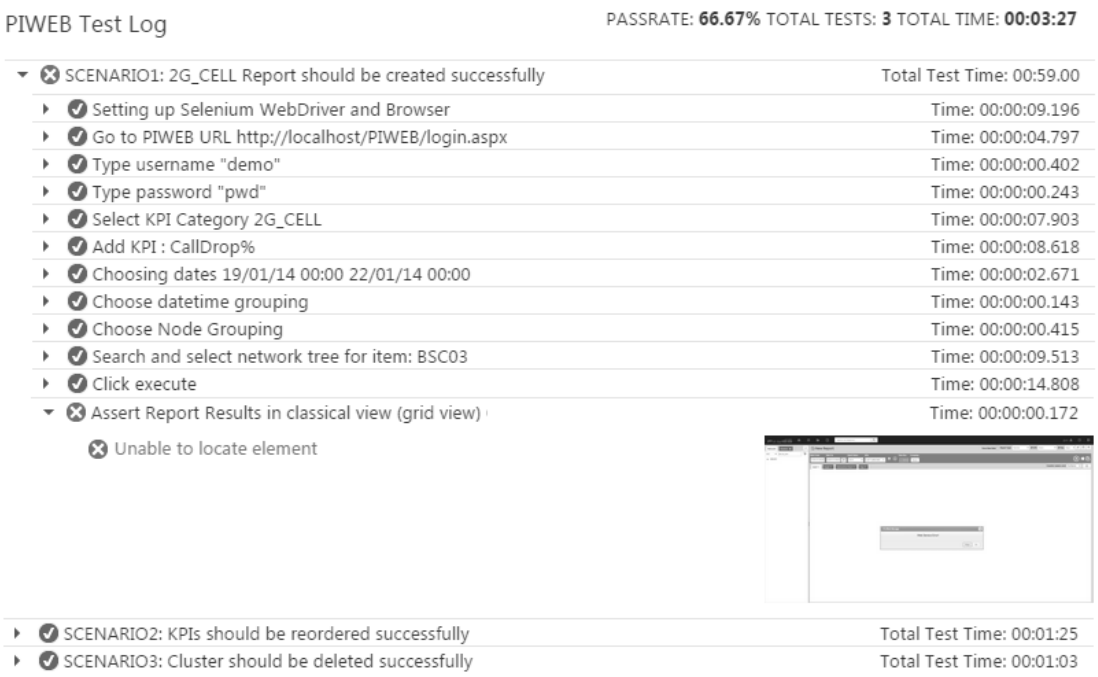

Figure 6: Sample output of PIATT.

CHAPTER VI

INDUSTRIAL CASE STUDY: NETWORK PERFORMANCE MONITORING SYSTEM

We evaluated our approach for the reproduction of field failures reported for the P.I.Web system. 160 failure scenarios were successfully reproduced by automated processing of the reported issues, which account for 40% of the reported failures. As a result, we observed significant reduction of the time and effort for analyzing issue reports. We also proved the potential of issue reports as a source that can be exploited to automatically generate and execute test cases.

Many issues (60%) still required manual analysis by the software developers due to ill-formed, incomplete issue reports. We wanted to see to what extent training of the QA team increases the proportion of issue reports that can be automatically processed, if it does at all. Training sessions were performed to improve the quality of issues created by the QA Team consisting of 8 test engineers, 3 of them being at the senior level and the other 5 at the junior level. Training sessions were held for 3 days and they took 10 hours in total.

Table 3: The number of issue reports that are analyzed by clarity of step definition before and after the training session.

Issues by description	**Before T.**	**After T.**
Issues with defined steps	227	398
Issues without defined steps	430	244
Total:	657	642

Table 3 lists the overall results. In total 642 issues were compatible with the version we worked on and 398 of them were clearly described with the necessary steps to reproduce failures, which were 657 and 227 before training respectively [1].

[1]The number of compatible issues were decreased due to a major version upgrade.

Figure 7 represents the trend of issues that could be reproduced automatically. For 30 weeks, we have recorded the number of issues that can be reproduced (compatible issues) and the number of issues that can not be processed (not compatible). Figure 7 shows these numbers per week. It can be observed in general that, the number of compatible issues are more than the incompatible ones. Before major version change on Week 44 on 2014, it was possible to reproduce most of the issues. In several other days, there exists acceptable number of compatible issues, although the number of incompatible issues increase due to several reasons described in Section 7.

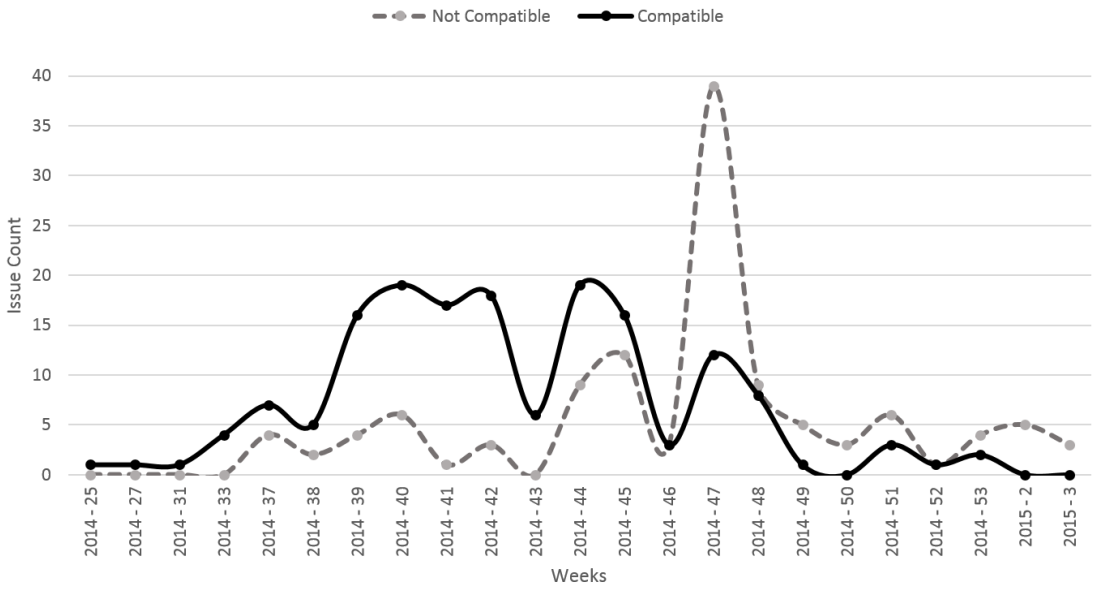

Figure 7: Issue compatibility trend for auto-reproducibility.

We also identified issues that can be automatically reproduced by PIATT compatibility. Results are listed in Table 4. We saw that 40% of the all issues could be converted after the training session, while this number was only 19% before training. These results show that significant improvements can be achieved with a relatively short training session. It takes approximately 10 minutes on average to reproduce a reported failure manually while debugging. Thus, our approach can lead to significant time and effort reduction.

Table 4: The number and ratio of issue reports with defined steps that can be used for automated failure reproduction before and after the training session.

Issues by PIATT compatibility	**Before T.**	**After T.**
Issues reproducible by PIATT	43 (19%)	160 (40%)
Issues not reproducible by PIATT	184	238
Total:	227	398

In total, 238 issue reports were not usable. That is, our approach failed to reproduce the reported failures in these issues. In the following chapter, we discuss the causes for this and the limitations of our approach.

CHAPTER VII

DISCUSSION

We were able to generate test cases and reproduce failures for many issue reports. However, there were also many cases, where our approach failed to utilize issue reports for automated failure reproduction. There are several causes for this. In this section, we discuss these causes and the sources of limitation for our approach.

We have identified four main causes: 1) lack of mandatory information in the issue description, 2) failure scenario complexity, 3) GUI design problem or a user action that is not performed via the browser GUI, 4) unimplemented test step methods.

Table 5 lists the causes and the corresponding number of issues that were unusable because of the listed causes. We realized that 11 of the issues were lacking information in terms of automation. That is, a person can reproduce the issue manually by supplying complementary information intuitively. In addition, 47 of the issues were too complex to implement as automated test cases. 61 failure scenarios could not be reproduced due to the involvement of out-of-browser actions such as database or command line actions or "*GUI Structure and Aesthetics Fault*" [22] which cause the components not be used by Selenium.

We observed that the majority of the unusable issues (119) could be used for failure reproduction if the library of scenario templates were extended with the

Table 5: The number of unusable issues per cause of unusability.

Cause	**Issue Count**
Lack of mandatory information	11
Failure scenario complexity	47
GUI problem / cannot be replayed on browser	61
Lack of scenario template or test step	119
Total:	238

relevant test steps. Developers can be notified for such of missing templates and test steps to improve the efficiency of the approach.

A potential limitation of our approach is GUI fragility. Major changes in GUI may require use case templates to be fixed in each GUI revision. This can be handled partially by creating design rules to preserve alignment of GUI and the automation tool.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

We introduced an approach for automatically reproducing field failures to aid their debugging. Our approach uses semi-formal failure scenario descriptions provided by system users and test engineers. These descriptions are utilized for automatically generating and executing test cases to reproduce failures. We evaluated the approach in the context of an industrial case study from the telecommunications domain. We observed significant reduction of time and effort for analyzing issue reports as a result of the application of our approach. Issue reports that are ill-formed, incomplete or irrelevant, have to be manually analyzed by software developers in any case. However, many reports can be automatically converted to reusable test cases. In our case study, 40% of all failure scenarios were successfully reproduced by automated processing of the reported issues. In addition, we proved the potential of issue reports as a source that can be exploited to automatically generate and execute test cases.

As future work, we foresee the following two opportunities and possible extensions regarding our approach. First, advanced natural language processing techniques can be applied on parsed issue reports to calculate similarity among them. By this way, similar issue reports can be grouped and related to each other automatically. This automated categorization can further reduce the analysis time for developers. Second, system users and test engineers can be guided by tools to define well-structured issue reports. These tools can parse issue reports during their creation and they can provide immediate feedback for missing information and structural flaws. This guidance can improve the quality of issue reports and increase the proportion of reports that can be automatically converted to test cases.

APPENDIX A

ILLUSTRATION

In this section, we illustrate an application scenario of the approach in the context of the P.I.Web case study.

Issues are initially reported through the ticketing system as shown in Figure 8 with problem definition as well as product related details like version, component or priority information. Such a report is very common in existing ticketing systems.

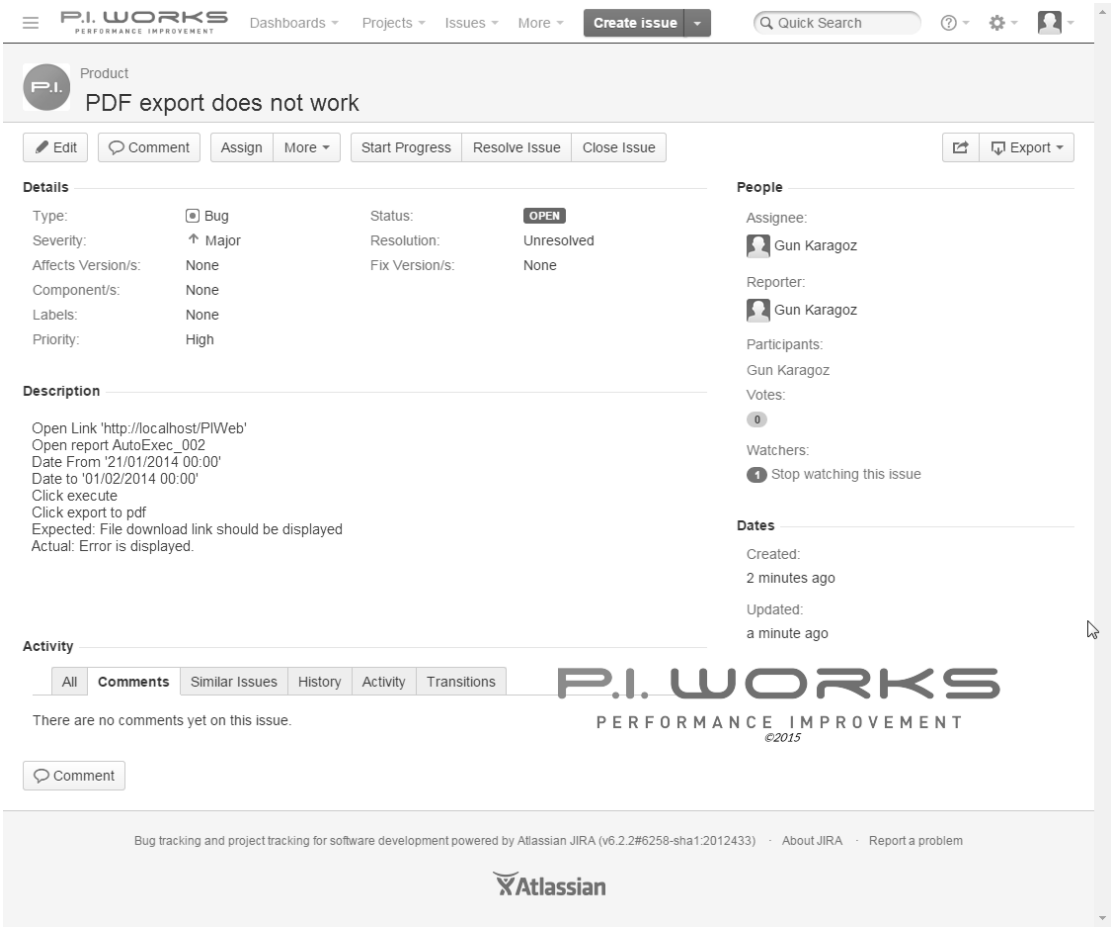

Figure 8: Issue creation on the ticketing system.

Then, the issue is converted to Gherkin format. Figure 9 represents the *BDD Parser* tool for parsing an issue description into Gherkin format.

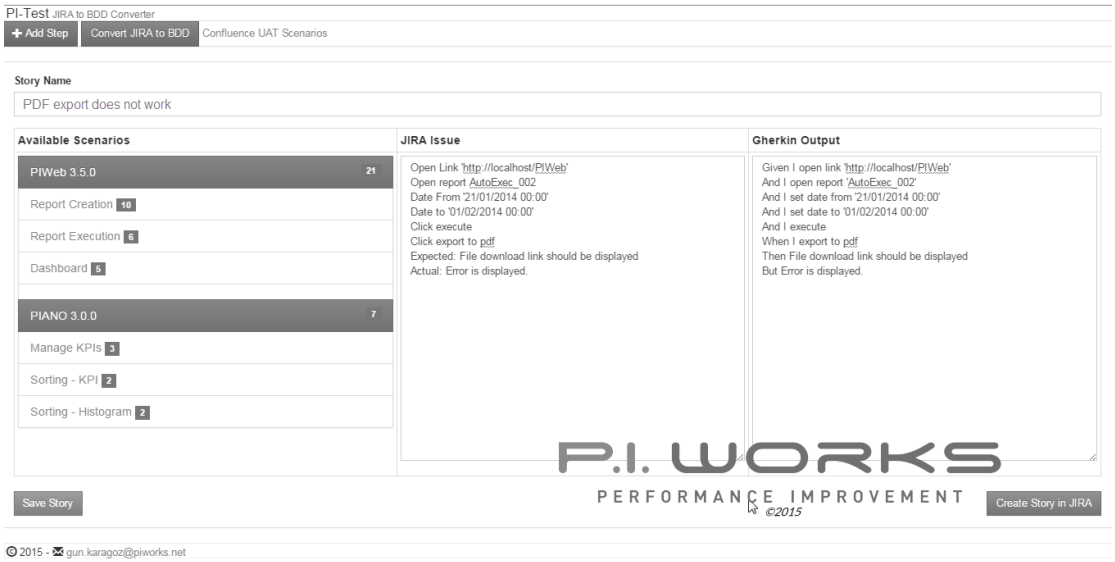

Figure 9: Issue conversion to the Gherkin format.

The converted issue description is used as input in the *Test Runner* module of the Test Automation Tool (Figure 10) for reproducing the corresponding failure scenario automatically.

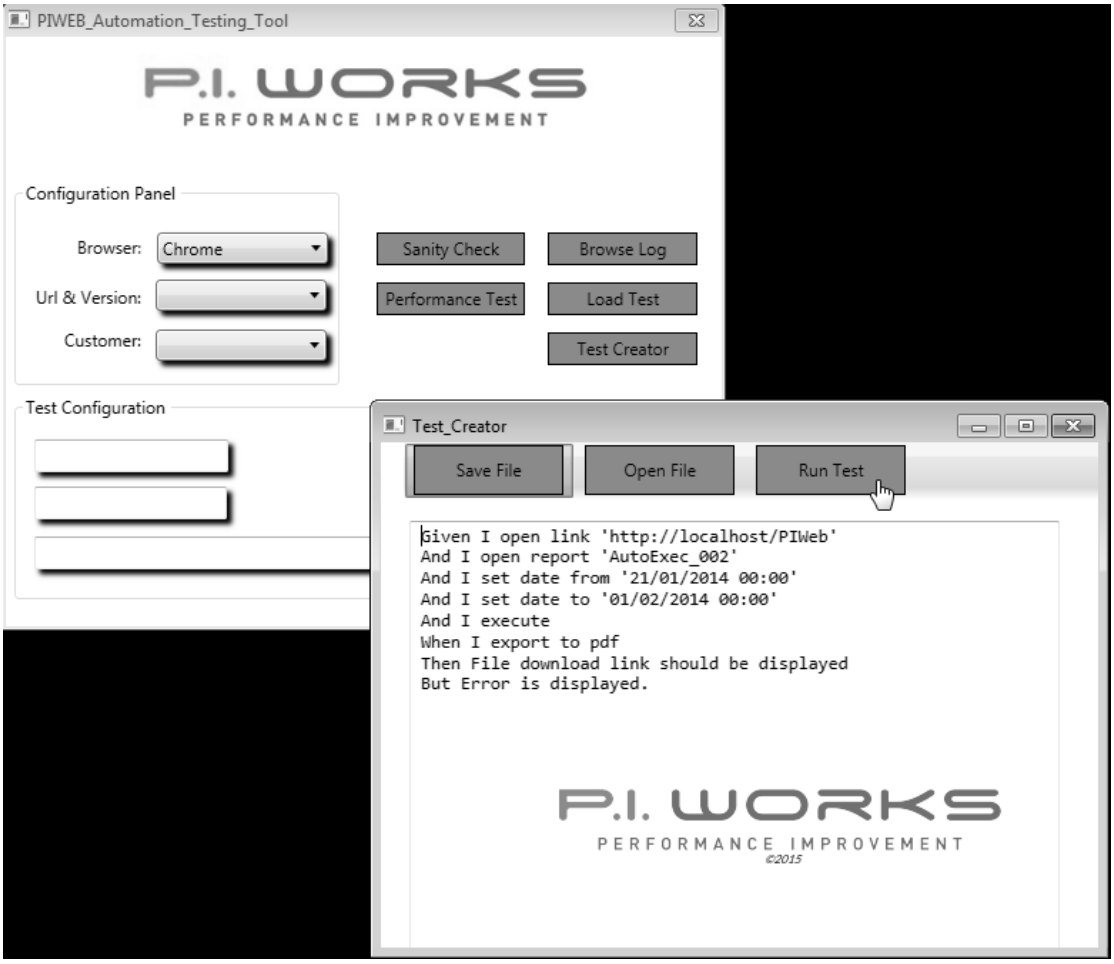

Figure 10: Addition of an issue as a test case to be executed by the test automation tool.

The *Test Runner* module replays user actions involved in the failure scenario based on the issue description in the given order. In the replayed set of user actions, a report is opened first (See Figure 11). Then, a group of report parameters are set such as the date range for the report.

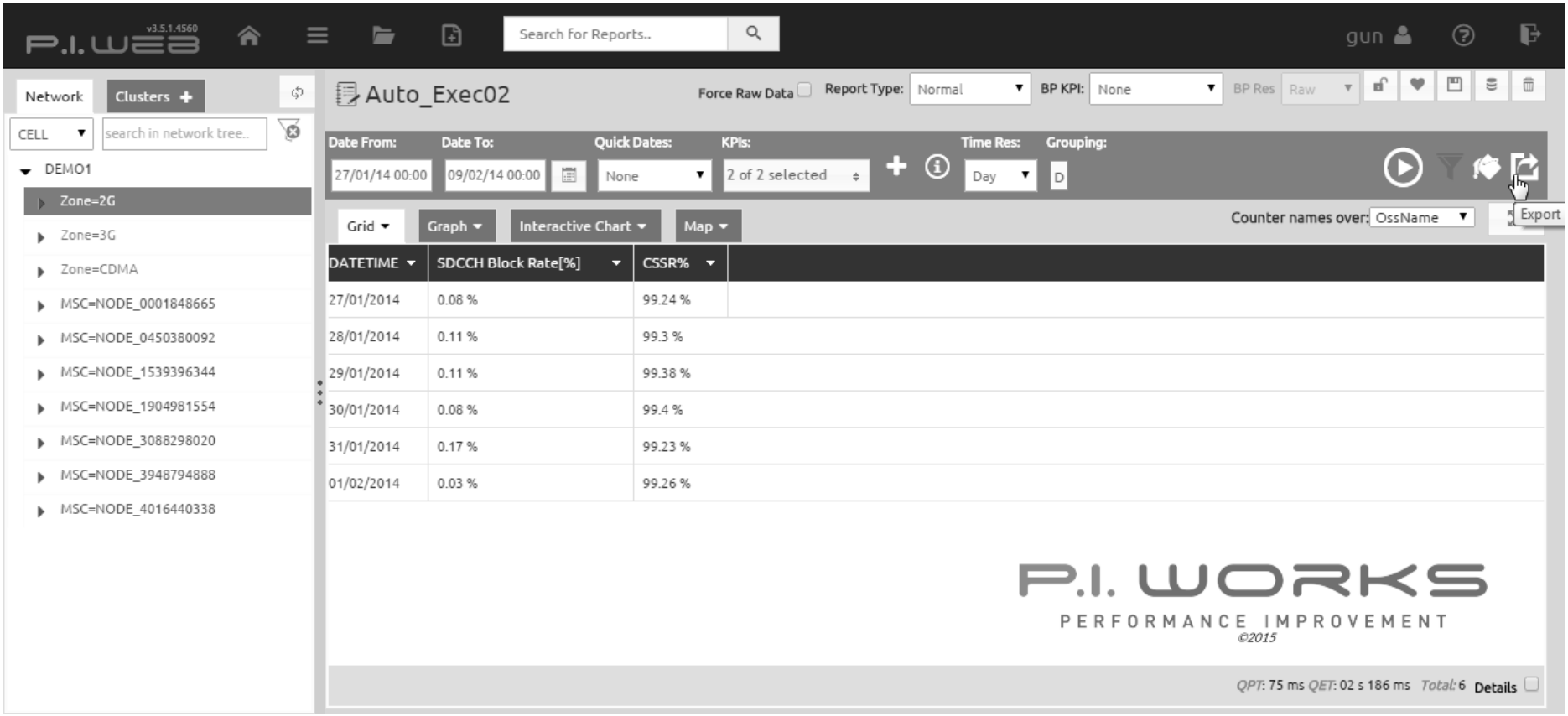

Figure 11: Report results for the given report.

The scenario proceeds with an *Export* action, which opens the report export options. Hereby, file download link creation fails as displayed in Figure 12, thus the *Download* button is not displayed on the user interface.

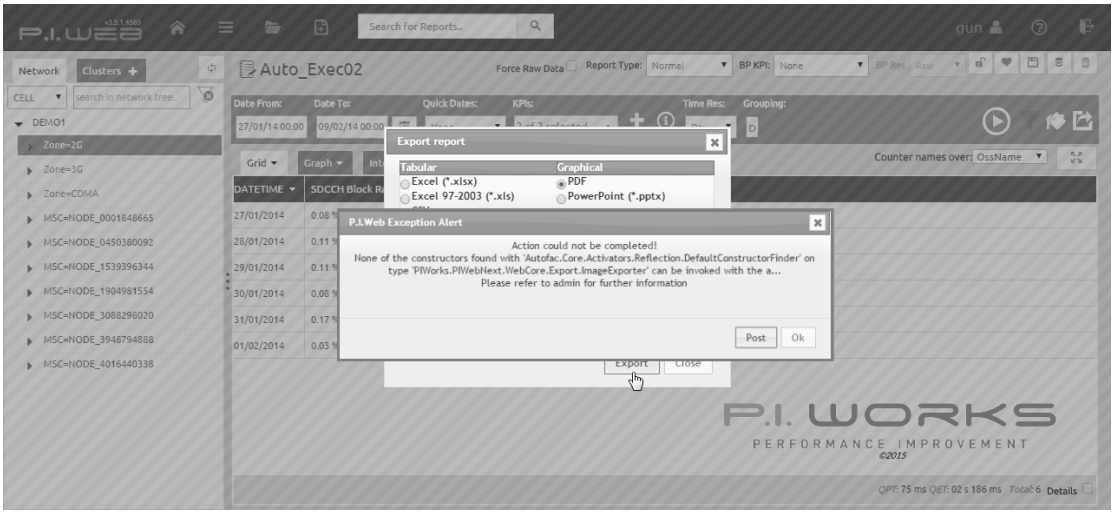

Figure 12: Export screen with fail message.

Test results are reported (Figure 13) together with the set of actions followed and a screen-shot of the screen at the time of failure.

PIWEB Test Log

PASSRATE: **0%** TOTAL TESTS: **1**

PDF export does not work	Total Test Time: 00:00:19
Open 'Google Chrome' web browser	Time: 00:00:00.000
Go to 'http://localhost/PIWEB'	Time: 00:00:01.567
Type username "****"	Time: 00:00:00.145
Type password "****"	Time: 00:00:00.150
Click on 'Login' button	Time: 00:00:00.079
Waiting... (Current page is 'in progress')	Time: 00:00:03.729
Wait until 'Logout Icon' is present on page	Time: 00:00:00.082
Wait until 'Search Report Box' is present on page	Time: 00:00:00.048
Searching for ' AutoExec_002 ' report	Time: 00:00:00.556
Waiting... (Current page is 'in progress')	Time: 00:00:00.603
Wait until 'Network Tab' is present on page	Time: 00:00:00.052
Click on 'Network' Tab	Time: 00:00:00.139
Wait until 'Network (in network tab)' is present on page	Time: 00:00:00.060
Click on 'Network' in 'Network' Tab	Time: 00:00:01.138
Waiting... (Current page is 'in progress')	Time: 00:00:00.031
Choosing dates between '21/01/14 00:00' and '01/02/14 00:00'	Time: 00:00:00.039
Wait until 'Execute Report' button is present on page	Time: 00:00:00.057
Click on 'Execute Report' button	Time: 00:00:00.476
Waiting... (Current page is 'in progress')	Time: 00:00:00.575
Wait until 'Execute Report' button is present on page	Time: 00:00:00.040
Click on 'Export' button	Time: 00:00:01.077
Select 'PDF' from selection list	Time: 00:00:01.061
Click on 'Export' button	Time: 00:00:01.079
Waiting... (Current page is 'in progress')	Time: 00:00:00.027
Click on 'Download' button	Time: 00:00:00.488

element not visible (Session info: chrome=42.0.2311.90) (Driver info: chromedriver=2.12.301325 (962dea43ddd90e7e4224a03fa3c36a421281abb7),platform=Windows NT 6.1 SP1 x86_64)

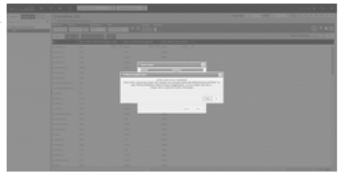

Figure 13: Test results report the failure regarding the reproduced issue.

After the bug is fixed by the developers, the same test case is executed on the new build. This time, test execution succeeds and the *Download* button becomes available as shown in Figure 14.

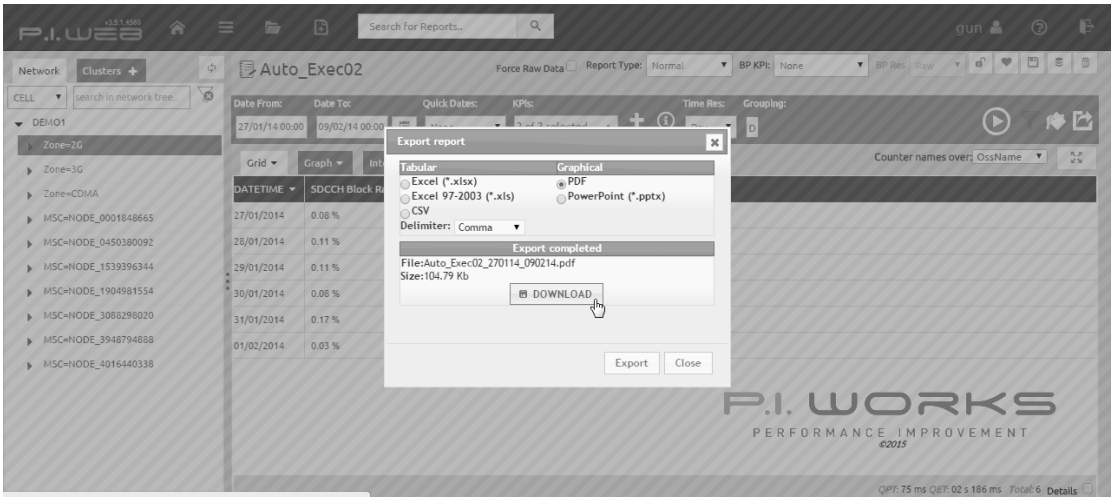

Figure 14: Export screen without fail message.

Bug fix is verified with a report (Figure 15) and the reported issue is closed.

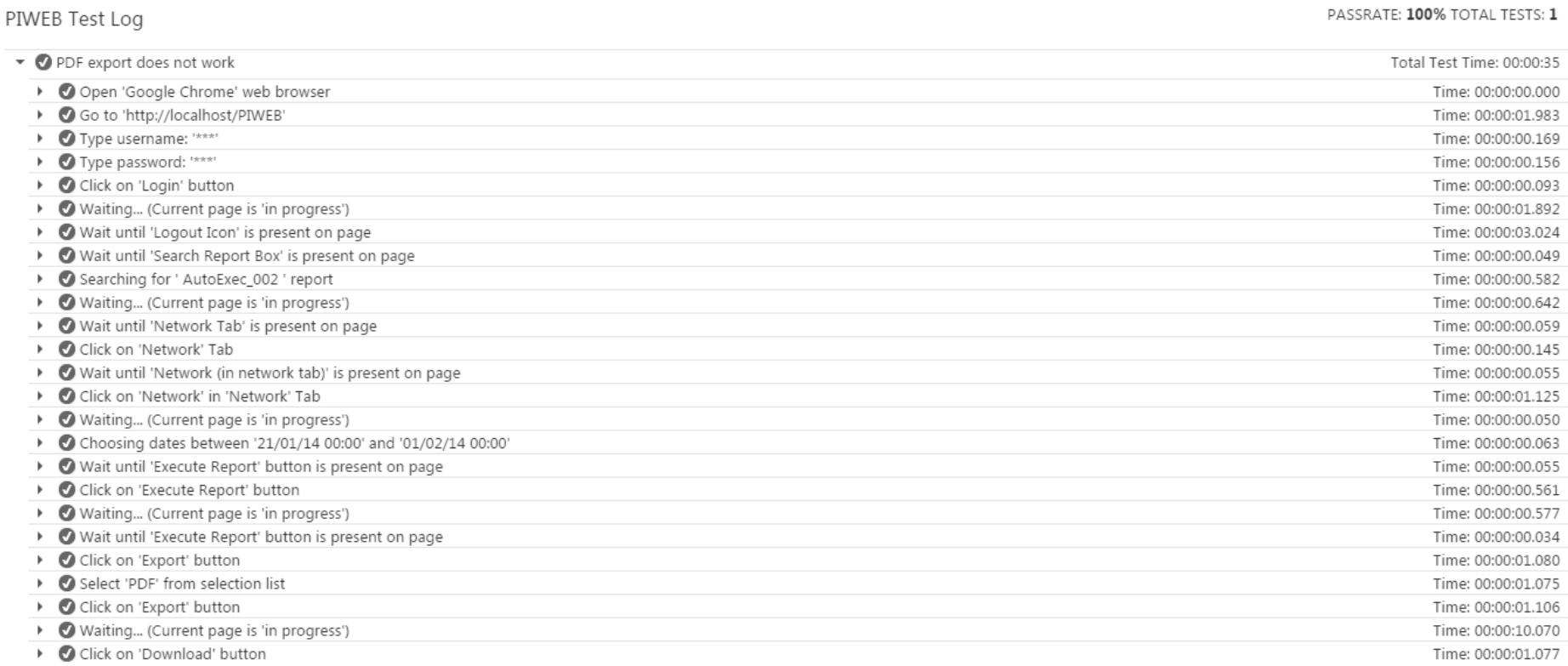

Figure 15: Test results report after the issue is fixed.

REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, 2004.

[2] K. Vikram, A. Prateek, and B. Livshits, "Ripley: automatically securing web 2.0 applications through replicated execution," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 173–186, 2009.

[3] J. Mickens, J. Elson, and J. Howell, "Mugshot: deterministic capture and replay for javascript applications," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pp. 159–174, 2010.

[4] S. Andrica and G. Candea, "Warr: A tool for high-fidelity web application record and replay," in *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 403–410, 2011.

[5] A. Bruns, A. Kornstadt, and D. Wichmann, "Web application tests with selenium," *IEEE Software*, vol. 26, no. 5, pp. 88–91, 2009.

[6] T. Roehm, S. Nosovic, and B. Bruegge, "Automated extraction of failure reproduction steps from user interaction traces," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pp. 121–130, March 2015.

[7] E. Laukkanen, M. V. Mäntylä, *et al.*, "Survey reproduction of defect reporting in industrial software development," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pp. 197–206, IEEE, 2011.

[8] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[9] M. Sarma and R. Mall, "Automatic test case generation from uml models," in *Proceedings of the 10th International Conference on Information Technology*, pp. 196–201, 2007.

[10] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[11] T. Pajunen, T. Takala, and M. Katara, "Model-based testing with a general purpose keyword-driven test automation framework," in *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 242–251, 2011.

[12] D. North, "Behavior modification," *Better Software Magazine*, pp. "27–31", 2006.

[13] J. Lerman, "Behavior-driven design with specflow," *MSDN Magazine*, vol. 28, no. 7, pp. 12–22, 2013.

[14] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, pp. 269–287, Springer-Verlag, 2012.

[15] N. Bajpai *et al.*, "A keyword driven framework for testing web applications," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 3, 2012.

[16] K. Arya and H. Verma, "Keyword driven automated testing framework for web application," in *Proceedings of the 9th International Conference on Industrial and Information Systems*, pp. 1–6, 2014.

[17] J. Hui, L. Yuqing, L. Pei, G. Shuhang, and G. Jing, "Lkdt: A keyword-driven based distributed test framework," in *Proceedings of the International Conference on Computer Science and Software Engineering*, vol. 2, pp. 719–722, 2008.

[18] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 474–484, 2012.

[19] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, "Reproducing field failures for programs with complex grammar-based input," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pp. 163–172, IEEE, 2014.

[20] K. Naveen and P. Hunter, "Cost effective agile test practices and test automation using open source tools specflow and white," in *Proceedings of the PNSQC*, pp. 165–178, 2012.

[21] J. Friedl, *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[22] V. Lelli, A. Blouin, and B. Baudry, "Classifying and qualifying gui defects," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pp. 1–10, IEEE, 2015.

[23] D. R. Kuhn, I. D. Mendoza, R. N. Kacker, and Y. Lei, "Combinatorial coverage measurement concepts and applications," in *Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 352–361, 2013.

[24] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *Proceedings of the AGILE Conference*, pp. 270–275, 2006.

[25] M. Sarma and R. Mall, "Automatic test case generation from uml models," in *Proceedings of the 10th International Conference on Information Technology*, pp. 196–201, 2007.

VITA

Gün Karagöz was born in 1987 in Ankara, Turkey. He received his B.Sc. degree from İzmir Institute of Technology in Electronics and Telecommunications Engineering in 2010. He joined Özyeğin University, Turkey in February 2011 where he worked as a teaching and research assistant until leaving on August 2012 for military duty. Until 2013, he served the Turkish armed forces. From 2013 until 2014, he worked as an Access Network Optimization and Planning Engineer at Turkcell, in Turkey. He is currently working as Product & UAT Senior Engineer at P.I.Works, in Turkey. During his master's studies, he focused on natural language processing, automatic speech recognition, software testing methodologies and test automation techniques.