University of Trento

Department of Information Engineering and Computer Science

Degree course in Computer Science

Final Thesis

Gherkin* and Cucumber*

A NEW TEST CASE PATH COMPOSITION APPROACH TO TESTING RUBY ON RAILS WEB APPLICATIONS

Supervisor:
Prof. Maurizio Marchese

Co-Supervisor:
Prof. Adolfo Villafiorita

Graduant:
Roberto Zen

Academic year 2013-2014

Ai miei genitori
A mio fratello

Acknowledgements

I would like to thank my supervisor Maurizio Marchese for his encouragement and support during the writing of this composition.

My work would have never been carried out without the help of the whole ICT4G Unit of Fondazione Bruno Kessler. In particular, I would like to thank Prof. Adolfo Villafiorita for his help and his patience with me during these last two years. You are a mentor for me.

Thanks also to Prof. Alberto Montresor for the useful discussion we had.

Thanks to my family for supporting me during my studies.

I want to sincerely express my gratitude and thanks to my best friends: Stefano, Sara, Sveva and Antonio.

I also acknowledge my roommates and friends: Victor, Damiano and Diego.

I would like also to thank all my friends, particularly Mirko Zaffaroni, Giovanni Bonetta, Andrea Sosi, Giovanni De Francesco, Giulio Fornasaro, Luca Zamboni, Amedeo Calafiore, Andrea Balzan, Chiara Salvagno, Lucia Pilat, Anna Giamosa and Federica Stetka.

Contents

List of Figures

List of Tables

Abstract

Software testing is one of the most important phases of software development processes. Its main goals are: finding bugs, meeting the needs of stakeholders and proving that each part of software works as expected.

Great tests can be written by testers with experience, intuition and also a lot of creativity. One way to measure the adequacy of tests is called code coverage. It represents the percentage of lines of code covered by executed tests. One goal of testers is reaching the 100% of code coverage or arrive close to it. However, having the maximum percentage of coverage on a project gives a false sense of confidence. In other words, if part of the software is tested with 100% of coverage, nothing can be guaranteed about its robustness. In fact, test cases usually cover only the logical workflow of the entire software workflow. This problem limits considerably what we can infer with software testing.

This thesis presents a novel approach to testing Ruby on Rails web applications. This method automatically generates new test cases based on the composition of the single ones.

Chapter 1

Introduction

During the development of a software system it is important to test software itself and how it interacts with users. In this view, agile methodologies are adopted by software development teams due to their significant benefits. In some methodologies such as Scrum and Extreme Programming, one of the core activity done is Test Driven Development (TDD, [1]). This development technique is based on writing tests before developing enough production code, following the *Red-Green-Refactor* process. More precisely, for each functionality there should be some test cases written by testers. These tests will fail until the feature is developed, making the tests pass. In this way, new software features can be deployed only after they have been tested.

An extension of TDD is Behavior Driven Development (BDD, [2]) which is an agile software development technique evolved in the last few years. It uses natural language to describe acceptance so that tests that can be read by non-programmers. BDD focuses on writing user story specifications and then working on a single specification at time, until it is developed and tested. These specifications are called "scenarios" and each of these describes a possible example of how a software should work as a test case.

These two techniques are quite similar. The key difference is based on the terms *tests* and *scenarios*. In fact, while tests are written by developers, scenarios are given by stakeholders, written by testers, and implemented by developers.

The choice between TDD and BDD is sometimes complicated [3] but, to the best of my knowledge, BDD is the best approach due to two significant benefits:

1. Scenarios help testers to write test cases in their own language (i.e English). This allows developers to think through solutions, before starting to write code.

2. It is notoriously difficult to figure out exactly what stakeholders want, so an easy way is starting by examples. In this view, test cases can be read by stakeholders in order to understand if a new software feature meets the requirements.

However, this technique suffers of some issues. As the quantity of features in Web-based software systems continue to grow rapidly, the organization of the BDD test suite has become difficult for people who do not have a solid background on testing. Moreover, requirements changes may cause a complete refactoring of it and so, a labour-intensive process for testers.

Fortunately, Rails benefits from a huge number of "gems", which are packages that provides to developers self-contained solutions for various problems, also for testing. Two important gems in this field are *Gherkin* and *Cucumber*, which help testers to organize test cases as user stories.

Recently, many studies such as [4] and [5] have been carried out on the topic of test case generation for Web Applications. My research focuses on how this problem can be archived in Ruby on Rails web applications.

This thesis describes ***Gherkin**** and ***Cucumber**** Ruby gems, which help address some of the issues mentioned above. In particular, we designed a solution that may allow testers to better organize their tests and generate automatically new test cases.

I developed those gems during my internship at the ICT4G Unit[1] at Fondazione Bruno Kessler[2] (Trento, Italy).

The proposed solution aims to be a novel approach to testing Ruby on Rails web applications following the BDD methodology. It provides an easy way to automatically generate test cases starting from the composition of the single ones. Gherkin* allows testers to write scenarios which can be connected to each others. Cucumber* needs Gherkin* to parse files where scenarios are written, and builds a graph where nodes are scenarios while edges are links between them. Afterwards, a visit to that graph is run in order to execute all the possible test case paths in sequence.

To better understand how test case paths can be obtained, an analysis of graph coverage criteria has been done. Fundamental results have been found and will help to improve our solution in the near future.

[1]`http://ict4g.fbk.eu`

[2]`http://www.fbk.eu`

1.1 Motivations

The reasons which have motivated me to conduct a study on this field are the following:

1. **Having 100% coverage on a project gives a false sense of confidence** [6]. In fact, if a web application is tested with the maximum percentage of coverage nothing can be guaranteed about its robustness. It is necessary to find a method to improve the validation of web applications.

2. **Testers waste a lot of their time writing test cases**. Many test cases are very similar and consequently the same checks appear multiple times in different scenarios. This may be considered as a scalability problem.

3. **Test cases are written following testers' logic**. Lots of different situations are not tested because they may seem actions that users never do. This may be considered as a quality problem.

1.2 Goals

The goals of this thesis are the following:

1. **Investigate and experiment approaches to improve the validation of web applications**. This thesis aims to increase the percentage of code coverage of test cases written for web applications. Thanks to the composition and the generation of test case paths, some lines of code that have not been tested with written scenarios can be checked by the generated ones.

2. **Save testers' time**. The purpose of this elaborate is to avoid code duplication following the Don't Repeat Yourself (DRY) principle also for test cases.

3. **Improve the organization of test cases**. We want to organize test cases in small composable units to improve their maintainability and define composition rules that allow us to compose elementary test cases in complex scenarios.

1.3 Results

The tools developed as a result of this work can help to increase the validation of Ruby on Rails web applications in terms of code coverage. An extension of two existing Ruby gems, which are part of an open source project[3],

[3]`http://cukes.info`

will be presented. Gherkin* and Cucumber* can be installed and used by testers who want to use these new BDD tools in their web applications. Both the gems are currently released under the MIT license on my Github profile following the URLs: `https://github.com/robzenn92/gherkin_star` and `https://github.com/robzenn92/cucumber_star`.

1.4 Outline

Chapter 2 describes all the technologies we used to introduce the BDD in Bring The Food. Moreover, it gives readers all the competences to understand our proposed solution. Chapter 3 explains the Cucumber's limits and why testers find difficult to manage test cases with it. Chapter 4 shows the details about the proposed solution and how an ideal test workflow could be. The last chapter states the conclusion of the thesis.

Chapter 2

State of the art

2.1 Introduction

I started my internship as web developer and tester focusing on how TDD and BDD can be introduced and used during the *Bring The Food*[1] development process. Bring The Food is a crowd-sourcing web and mobile application that allows people to fight food waste. From now, I will refer to Bring The Food only as the Ruby on Rails web application.

Ruby on Rails[2] is an open-source web framework written in Ruby. It allows developers to build web pages and applications based on Ruby. Ruby on Rails emphasizes the use of well-known software engineering patterns and principles, such as active record pattern, don't repeat yourself (DRY), and model-view-controller (MVC). Rails was created in 2003 by David Heinemeier Hansson and has since been extended by the Rails core team and more than 3,400 contributors.

Ruby[3] is an open source programming language developed in the mid-1990s by Yukihiro "Matz" Matsumoto. It supports multiple programming paradigms, including functional, object-oriented, and imperative. Bring The Food is written in Ruby 1.9.3, the same version that I used to develop *Gherkin** and *Cucumber**.

2.2 Ruby and its approach to testing

As [7] states, a standard practice for project teams is to set up three exact and independent replicas of the same operating environment as follows.

- A **development environment** in which the development of applications take place. Developers usually use fancy data and work on them in order to make changes.

[1]`http://www.bringfood.org`
[2]`http://www.rubyonrails.org`
[3]`http://www.ruby-lang.org`

- A **testing environment** in which testers test if the system in the development environment is ready to be deployed.
- A **production environment** which is the version that final users use. Basically, when a new feature of an application is both developed and tested it will be deployed in this environment.

According to the official guide[4], every Ruby on Rails web application has a test environment with which testers can use to execute test cases, without changing anything about the other two. Furthermore, since Ruby 1.8, a TDD framework is available in every Ruby project by default. In our case, Ruby 1.9.3 provides *MiniTest*[5], a complete suite of testing facilities supporting TDD and BDD, which allows developers to easily set up, organize and run tests. With it, tests for models and controllers may be easily written without other external tools.

However, its syntax is not easy to read and maintain over time. Fortunately, there are a various of frameworks available online. These frameworks are called gems and they can be either downloaded directly using the RubyGems packaging system[6] or one of the world's largest code host: Github[7]. Gems help developers to build their web applications using facilities developed by other developers. In particular, there are some gems that help testers to write better tests. Most of the gems are released under the MIT License so, developers are free to use, copy, modify, or publish them. To solve the test code maintainability problem in Bring The Food we use *RSpec* with *Capybara*.

2.3 RSpec and Capybara

RSpec[8] is a BDD tool for Ruby programmers. It allows developers to write human readable specifications [8]. This means that it provides a syntax closer to users by using pseudo natural language (e.g., RSpec uses words such as *should* and *expect* rather than *assert*). Moreover, it takes a slightly different approach to the idea of testing applications, by testing "behavior" rather than only specific methods [9].

Capybara[9] helps testers to simulate how a real user would interact with them. For each test cases it opens an instance of a web browser and navigates between web pages. This gem is useful because with it testers are not only able to validate the behavior of web applications but also to test their graphic interface.

[4]`http://guides.rubyonrails.org/configuring.html`
[5]`https://rubygems.org/gems/minitest`
[6]`https://rubygems.org`
[7]`https://www.github.com`
[8]`https://rubygems.org/gems/rspec`
[9]`https://rubygems.org/gems/capybara`

These gems help us to write tests more quickly than MiniTest. Thanks to Capybara, we write tests which cover also the front-end side of the application. However, RSpec has one important limit: while for the BDD technique, tests should be understood by stakeholders, RSpec mixes test description with test code, making difficult to read them. To solve this problem, we use other two Ruby gems which are of particular importance: *Gherkin* and *Cucumber*.

Gherkin and Cucumber are development tools that allow testers to write *scenarios* [10]. They aims to bridge the gap between developers and stakeholders. These framework are both written in Ruby, but they can be used to test applications written either in Ruby or in other languages including Java, C# and Python. The strength of those gems is that testers write scenarios in natural language without worrying about Ruby code. In fact, they overcomes the RSpec limit by separating test code from acceptance tests (scenarios).

2.4 Gherkin

Gherkin[10] is the core library of Cucumber. It allows us to define test cases that testers write in files with the *.feature* extension. From now, these files will be called simply *feature files*.

While all the scenarios are written in these files, test code is written in separated Ruby files, which contain blocks of code called *step definitions*. These definitions are analogous to method definitions except one difference: their prototypes are defined by regular expressions.

The Gherkin gem allows Cucumber to parse feature files in order to execute all the scenarios written in them. An example of feature file taken from Bring The Food is shown in Figure 2.1.

```
Feature: Login and logout as collector

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"

  Scenario: Logout
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page
```

Figure 2.1: An example of a feature file taken from Bring The Food.

[10] https://github.com/cucumber/gherkin

As it can be seen, the syntax of feature files is really simple to understand since it uses a small set of keywords. As reported in the Cucumber Wiki[11], most lines in them start with one of the special keyword written in Table 2.1.

	Scenario Outline		
Name	*Native*	*Encoding*	*Scenario*
Feature	*Background*	*Examples*	*Given*
When	*Then*	*And*	*But*

Table 2.1: The set of Gherkin's keywords.

These keywords are used by testers during test cases definitions. In other words, testers combine keywords such as *Feature* and *Scenario* to define specifications for each new feature and new scenario to develop.

Every feature file consists of one single feature. In order to define a new feature, testers write the keyword *Feature* followed by a plain-text description. After the description, other text may be written in order to describe which tests will be covered in that feature. Each feature usually contains a list of scenarios, so testers may add the keyword *Scenario* on a new line. The text immediately following on the same line as the Scenario keyword is the name of the scenario. Each scenario contains a list of steps, each of these is composed by a sentence which must start by one of the words *Given*, *When*, *Then*, *And* and *But*. For Cucumber there are no differences between the previous keywords: all of them are treated as the same. However, those words should help testers to write descriptive test cases following the natural language. So, while the word "Given" should represent an initial condition, the word "When" should be interpreted as an action and the word "Then" should be considered as an expected result. The conjunctions "And" and "But" are usually used in case of multiple conditions, actions or results.

In addition to being a library, Gherkin defines the language that Cucumber understands. Consequently, its grammar defines how feature files must be written correctly.

To allow Cucumber to parse feature files, Gherkin defines a parser using Ragel[12], which is a parser generator that allows users to define a parser starting from regular expressions. The parser is written starting from the grammar, which is defined by productions written as regular expressions in a bottom-up notation. The flow is the following: Gherkin uses the grammar's file as input to Ragel which compiles it into a Finite State Machine (FSM) and produces the final parser as output in a Ruby file. Afterwards, the

[11]`https://github.com/cucumber/cucumber/wiki/Feature-Introduction`
[12]`http://www.ragel.org`

parser uses a parsing table to check if a feature file is syntactically correct. Table 2.2 shows the parsing table used by Gherkin. As can be seen, most of the cells in the table contain the Expected Error (E) value. The parser occurs these cells when a file violates a rule of the language. In this case, a parsing error is raised and the parsing process is stopped.

	feature	background	scenario	scenario_outline	examples	step	row	doc_string	eof
root	feature	E	E	E	E	E	E	E	eof
feature	E	background	scenario	scenario_outline	E	E	E	E	eof
step	E	E	scenario	scenario_outline	E	step	step	step	eof
outline_step	E	E	scenario	scenario_outline	examples	outline_step	outline_step	outline_step	eof
background	E	E	scenario	scenario_outline	E	step	E	E	eof
scenario	E	E	scenario	scenario_outline	E	step	E	E	eof
scenario_outline	E	E	E	E	E	outline_step	E	E	eof
examples	E	E	E	E	E	E	examples_table	E	eof
examples_table	E	E	scenario	scenario_outline	examples	E	examples_table	E	eof
eof	E	E	E	E	E	E	E	E	E

Table 2.2: The parsing table used by Gherkin to parse feature files.

Consider now that some keywords shown in Table 2.1 are not used for defining test cases. Moreover, even though there are slight differences between *Scenario* and *Scenario Outline*[13], both keywords define a test case. Hence, we can focus on the first word, and in particular on a set of keywords which is smaller than the original one.

$$\{\, Feature, Background, Scenario, Given, When, Then, And, But \,\}$$

The set defined above includes all the keywords that can be used by testers in order to define test cases. The keyword *Background*, is used to define conditions that must be satisfied at the beginning of each scenarios written in the same file. Since all the step keywords *Given*, *When*, *Then*, *And* and *But* are treated as the same by Cucumber, we can group all of them in one set we call Step, obtaining the following result.

$$\{\, Feature, Background, Scenario, Step \,\}$$

Figure 2.2 shows part of the FSM produced by Ragel. In that automaton, all the states are necessary to parse feature files composed by the keywords above. The same automaton can be obtained considering Table 2.2 as follows: while the first column contains all the states of the automaton, the first row shows all the possible tokens that can be recognized by the parser and the remaining cells are transitions between states. There is an initial state *root* and a final state *eof*. All the transitions that outcome or move back to the E state are not considered. A feature file is correct only if the parser arrives in the eof state once that the file is completely parsed. For

[13]`https://github.com/cucumber/cucumber/wiki/Scenario-Outlines`

instance, if we consider the *Feature* state we can note that there are three transitions outcome from it. These transitions are allowed because after a *Feature* definition, testers may define either a new *Scenario* or a *Background* definition. However, they can not define directly a step definition because steps must be defined within a *Scenario* or a *Background*.

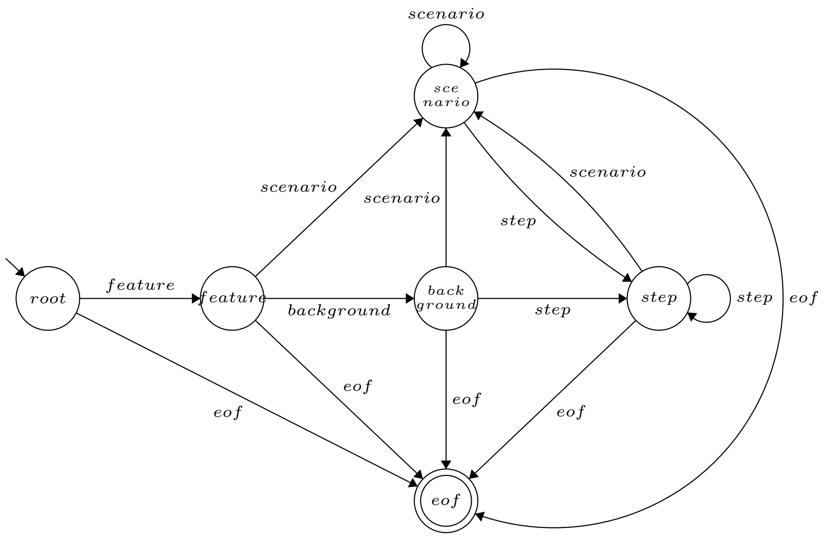


Figure 2.2: The FSM built by Ragel.

Consider now, that every state in the automaton shown in Figure 2.2 has a transition that goes to the final state. Hence, we can reduce the graph removing the *eof* state and converting all the states in finals. The result can be seen in Figure 2.3.

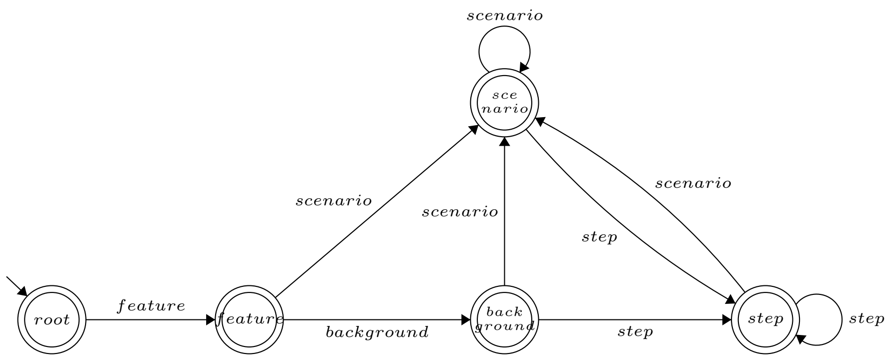


Figure 2.3: A simplified version of the FSM built by Ragel.

While the parsing process is running, the parser sends an event to a Cucumber listener which executes semantic actions for each keyword found.

2.5 Cucumber

Cucumber[14] uses Gherkin to parse feature files in order to execute all the scenarios.

Every time that the Gherkin's parser finds a keyword, it sends to the Cucumber listener that word followed by the plain-text written until the next keyword. This mechanism is made possible because keywords are always followed by plain text (i.e, see Figure 2.1). When these values are received by the listener, Cucumber runs a semantic action associated to that keyword. Basically, these actions are functions that build an Abstract Syntax Tree (AST) for each feature file analysed. In particular, every node in the tree has the structure illustrated by Figure 2.3.

parent	
keyword	*plaintext*
$child_1$ ... $child_n$	

Table 2.3: The structure of nodes in the AST built by Cucumber.

The tree is built as following: when the keyword *Feature* is recognized by Gherkin, a new instance of an AST is created and a new node composed by the word Feature and its description is added as root to that tree. When the word recognized is *Scenario*, a new child node is added to the root node. This node will contain the values Scenario as keyword and the name of the scenario as plain-text. The same process happens when a step keywords is found: all the steps of a scenario becomes children nodes of that scenario node. Steps will be leaf nodes of the tree.

Figure 2.4 shows the AST built by Cucumber for the feature file shown in Figure 2.1.

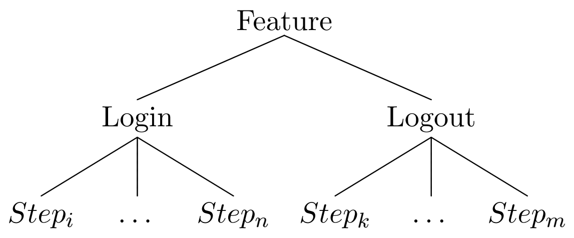


Figure 2.4: An example of an Abstract Syntax Tree built by Cucumber.

In the end of the parsing process, Cucumber has an AST instance with the same structure of the feature file parsed.

[14]`https://github.com/cucumber/cucumber`

To execute all the scenarios stored in the AST, Cucumber creates an instance of a *Visitor* class which starts a preorder tree traversal. Each scenario is executed as a separate test case, with no interaction to other scenarios or features. The output produced by Cucumber is the following:

- If the node visited has the words Feature or Scenarios in the keyword field, the output is the keyword followed by the plain-text as written in the input file.
- If the node visited has a step keyword in the keyword field, Cucumber tries to match the plain-text with a regular expression written in a step definition file. If a match is found then the test code written into that step definition is executed, otherwise a pending error is raised and the execution is stopped.

Cucumber also formats the output on the terminal providing indentation and displaying each step with a color: passing steps are coloured green, failing steps red, and undefined and pending steps yellow. When a step fails or is undefined, Cucumber skips to the next scenario and colors the skipped steps cyan[15].

The output of the Cucumber execution of the feature file visible in Figure 2.1 can be seen in Figure 2.5.

```
Feature: Login and logout as collector

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"

  Scenario: Logout
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page

2 scenarios (2 passed)
6 steps (6 passed)
0m0.010s
```

Figure 2.5: The result of the execution of Cucumber.

[15]`https://github.com/cucumber/cucumber/wiki/Step-Definitions`

2.6 SimpleCov

To gather information about our tests in Bring The Food we use a code coverage analysis tool for Ruby called *SimpleCov*[16]. SimpleCov gathers code coverage data from all tests including Cucumber's step definitions, RSpec and Capybara. It analyses which lines of code are covered by tests and then formats the results providing an html output file where the percentage of code coverage can be seen for each Ruby file in the project. We can also see the coverage for categories such as controllers, models, mailers and helpers.

The first time we used SimpleCov in Bring the Food we got the 50% of code coverage. This result was expected considering the following aspects:

1. We tested only the main features of the application including the Sign up, the Login, the Logout and other actions such as publishing, booking, editing and deleting food donations.

2. We introduced the BDD technique when most of the application's features were already developed.

If we consider the results provided by this tool, we may infer that the 50% of our application has been tested. Unfortunately, in a high-level language like Ruby, some lines of code marked as covered by SimpleCov, may not be tested by test cases. In fact, some of these are marked simply by loading the test environment.

2.7 Gems overview

All the previous gems can be combined in order to enforce the validation of Ruby on Rails web applications.

SimpleCov
Gherkin and Cucumber
RSpec and Capybara
MiniTest

Table 2.4: The BDD gem stack used in Bring The Food.

Table 2.4 shows how I used those gems: starting from the bottom, where the default MiniTest framework can be seen, going up to the last gem introduced in the Bring The Food development process.

[16]`https://rubygems.org/gems/simplecov`

Chapter 3

The limits of the Cucumber gem

To reach a satisfactory level of test coverage, and obtain an high percentage value in SimpleCov, testers need to write lots of test cases for each feature. However, managing lots of tests in a single file may be difficult. Consider also that, scenarios may be similar to each other. Hence, the same steps often occur in different scenarios.

For instance, consider the Login condition: most of our scenarios use it to test features that require a user which is logged in the application. Consequently, also same results can be expected by different test cases (e.g., Login failed with a wrong username, wrong password, wrong username and password). As result, in order to validate little changes, tests repeat other tests or part of them more than once. To better understand this situation, we can define three categories of steps:

1. **Category 1**: conditions written to prepare an environment which will be the object of the testing activity. These steps usually start with the word *Given*.

2. **Category 2**: steps used to test the behavior of features in particular scenarios. These steps are usually written in the middle of test cases and start with the word *When*.

3. **Category 3**: checks which are used by Cucumber to validate test cases. These steps usually start with the word *Then*.

Every time Cucumber runs a test, steps that belong to both the first and the third category are always defined and executed by many different test cases testing similar functions. Therefore, those steps violate the DRY principle in Ruby, so our goal is to avoid duplicate of them. This provides us with a credible basis to believe that this situation is due to the fact that each scenario is run as a separate test case.

3.1 A real example

A clear example of the problems mentioned in the previous section is shown in Figure 3.1.

```
Feature: Example of Collectors' actions

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"

  Scenario: Logout
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page

  Scenario: Login and Logout
    Given I am on the login page
    When I sign in as "collector@example.com"
    And I sign out
    Then I should be redirected to the login page

  Scenario: A collector creates a donation
    Given I am logged in as "collector@example.com"
    When I publish a new donation
    Then I should see the last donation in the list of donations

  Scenario: A collector creates two donations
    Given I am logged in as "collector@example.com"
    When I publish two new donations
    Then I should see the last 2 donations in the list of donations

  Scenario: A collector creates a donation and then logs out
    Given I am logged in as "collector@example.com"
    When I publish a new donations
    And I sign out
    Then I should be redirected to the login page
```

Figure 3.1: Some scenarios repeat steps that has been already executed.

As it can be seen, there are six scenarios that test the following features: login, logout, and the creation of a new donation. In particular, we can note that three of them expect the same page redirect, while four of them have the same initial condition: the login of the user as collector.

Therefore we believe that is crucial to give testers a method which follows the DRY principle allowing an easier and a faster way to write and organize test cases. The solution we found is to extend the Gherkin and

Cucumber gems in order to limit step repetitions and to generate test case automatically.

3.2 Related work

Many approaches have been proposed to address the problem of test case generation.

In [11], a method for automatic testing based on an extended IA (EIA) model is provided, while [4] presents a two-phase approach to generate test cases automatically by analyzing structure of Web applications.

One approach we found similar to our work is [12] because it faces the same problem in terms of Ruby on Rails Web applications. It sketches an algorithm for generating test cases based on uncovered lines of code. To archive this, it automatically writes new step definitions in feature files. In this way, testers write only test code to make pass the steps.

However, our approach is different because the proposed solution allows testers to generate and execute multiple scenarios based on the composition of single test cases.

Chapter 4

The proposed solution

4.1 Gherkin*

To allow testers to write test cases that can be automatically executed, it is necessary first to extend Gherkin. We developed *Gherkin**, which extends the Gherkin language and adds the possibility to link scenarios each others.

Consequently, testers could write some scenarios to test initial conditions and others to test final results. Then, they could write scenarios that test specific behaviour of some features and connect all of them together.

*Gherkin** adds the new word ***Next_Scenario*** to the original set of keywords (see Table 2.1). The idea is the following: we want to allow testers to write the keyword *Next_Scenario* in order to link test cases and avoid step repetitions. This keyword aims to be a link between two test cases and it will be used by *Cucumber** (see the next section) to execute two or more scenarios in sequence.

If we consider the feature file shown in Figure 2.1, the first problem we face is to understand where that keyword may be written respect to the others. In order to solve this issue, we need to define some rules that testers must follow, so we extend the original grammar.

The process of making changes to the Gherkin's language involves little changes to its grammar. However, a grammar extension is not always an easy task: it may contain lots of productions and several terminal symbols. Hence, it can be difficult to find the right place to edit. Moreover, by adding other productions we may generate a completely different language, making the grammar ambiguous or recursive.

To avoid these situations and to validate the new grammar we write tests. There are a lot of scenarios written into the test folder of the Gherkin gem. We use them and we add more tests that check the correctness of the new productions as follows. Given a feature file as input to a scenario, the goal is to check that no parser errors occur for that file.

The first test we write leads us to validate the basics use of our solution.

More precisely, we want to test that a *Next_Scenario* definition may appear after the last step of a *Scenario*. Hence, we write the feature file shown in Figure 4.1 and we run the test.

```
Feature: Login and logout as collector

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"
  Next_Scenario: Logout

  Scenario: Logout
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page
```

Figure 4.1: An example of a feature files.

Obviously, the test fails because the original grammar does not contain productions for the new keyword. To pass the test, we first add some productions to the grammar. Those productions allow the parser to successfully parse new lines that contain the *Next_Scenario* token followed by a name of one scenario. As we already said, those productions leads also the parser to send an event to the Cucumber*'s listener with both the keyword and the plain-text values parsed in that line.

However, the parser uses also a parsing table to parse feature files, so we need also to extend that table: we add a new row and a new column, both start with the *next_scenario* word. The highlighted part of the Table 4.1 shows the first changes applied to the original parsing table. Consider that the position of the new row and the new column is not relevant.

	feature	background	scenario	scenario_outline	next_scenario	examples	step	row	doc_string	eof
root	feature	E	E	E		E	E	E	E	eof
feature	E	background	scenario	scenario_outline		E	E	E	E	eof
step	E	E	scenario	scenario_outline		E	step	step	step	eof
outline_step	E	E	scenario	scenario_outline		examples	outline_step	outline_step	outline_step	eof
background	E	E	scenario	scenario_outline		E	step	E	E	eof
scenario	E	E	scenario	scenario_outline		E	step	E	E	eof
scenario_outline	E	E	E	E		E	outline_step	E	E	eof
next_scenario										
examples	E	E	E	E		E	E	examples_table	E	eof
examples_table	E	E	scenario	scenario_outline		examples	E	examples_table	E	eof
eof	E	E	E	E		E	E	E	E	E

Table 4.1: The first changes applied to the original parsing table of Gherkin*.

Once the parsing table contains the definition of the *Next_Scenario* keyword, it is necessary to understand which values should be written into the highlighted cells. As we already explained in Section 2.4, each cell define a rule that must be obeyed by testers. More precisely, each cell represent a transition in the FSM produced by Ragel.

If we consider the feature file shown in Figure 4.1, the new parsing table should have a rule that allow testers to write the new keyword after the last step of the first scenario. We add the keyword *next_scenario* in the cell corresponds to the row *step* and the column *next_scenario*. Furthermore, we do not want that the opposite way happens, allowing therefore *step* definitions after a *Next_Scenario* line. This situation can be avoided by adding an Expected Error (E) value in the cell which cross the row *next_scenario* and the column *step*. Moreover, to allow that a *Next_Scenario* definition can be followed by a *Scenario*, it is necessary to add one more rule in the parsing table. So, we add the keyword *next_scenario* in the cell corresponds to the row *next_scenario* and the column *scenario*. The results can be seen in Table 4.2.

	feature	background	scenario	scenario_outline	next_scenario	examples	step	row	doc_string	eof
root	feature	E	E	E		E	E	E	E	eof
feature	E	background	scenario	scenario_outline		E	E	E	E	eof
step	E	E	scenario	scenario_outline	next_scenario	E	step	step	step	eof
outline_step	E	E	scenario	scenario_outline		examples	outline_step	outline_step	outline_step	eof
background	E	E	scenario	scenario_outline		E	step	E	E	eof
scenario	E	E	scenario	scenario_outline		E	step	E	E	eof
scenario_outline	E	E	E	E		E	outline_step	E	E	eof
next_scenario			scenario				E			
examples	E	E	E	E		E	E	examples_table	E	eof
examples_table	E	E	scenario	scenario_outline		examples	E	examples_table	E	eof
eof	E	E	E	E		E	E	E	E	E

Table 4.2: The details of a rule in the parsing table of Gherkin*.

These three rules are necessary to allow testers to write a feature file like the one shown in Figure 4.1. However, to check if the parser can parse that file, we run another time all the tests, and this time they pass. Afterwards we introduce other rules that allow feature files to have multiple *Next_Scenario* definitions within the same Scenario. An example use is login scenarios which are used not only to check if the login feature works as expected but also as conditions for some other scenarios or features. We also allow testers to write the new keyword in *Scenario Outline* contexts. The reason, as we already said, is that *Scenario* and *Scenario Outline* are test case so in the final parsing table shown in Figure 4.3 we can see that there is a rule for both. The *eof* value in the last column allow the word *Next_Scenario* to be the last keyword in a feature file. In all the remaining cells we set the value *E*.

	feature	background	scenario	scenario_outline	next_scenario	examples	step	row	doc_string	eof
root	feature	E	E	E	E	E	E	E	E	eof
feature	E	background	scenario	scenario_outline	E	E	E	E	E	eof
step	E	E	scenario	scenario_outline	next_scenario	E	step	step	step	eof
outline_step	E	E	scenario	scenario_outline	next_scenario	examples	outline_step	outline_step	outline_step	eof
background	E	E	scenario	scenario_outline	E	E	step	E	E	eof
scenario	E	E	scenario	scenario_outline	next_scenario	E	step	E	E	eof
scenario_outline	E	E	E	E	next_scenario	E	outline_step	E	E	eof
next_scenario	E	E	scenario	scenario_outline	next_scenario	E	E	E	E	eof
examples	E	E	E	E	E	E	E	examples_table	E	eof
examples_table	E	E	scenario	scenario_outline	E	examples	E	examples_table	E	eof
eof	E	E	E	E	E	E	E	E	E	E

Table 4.3: The parsing table used by Gherkin*.

Consequently, also the FSM produced by Ragel must respect all those new rules.

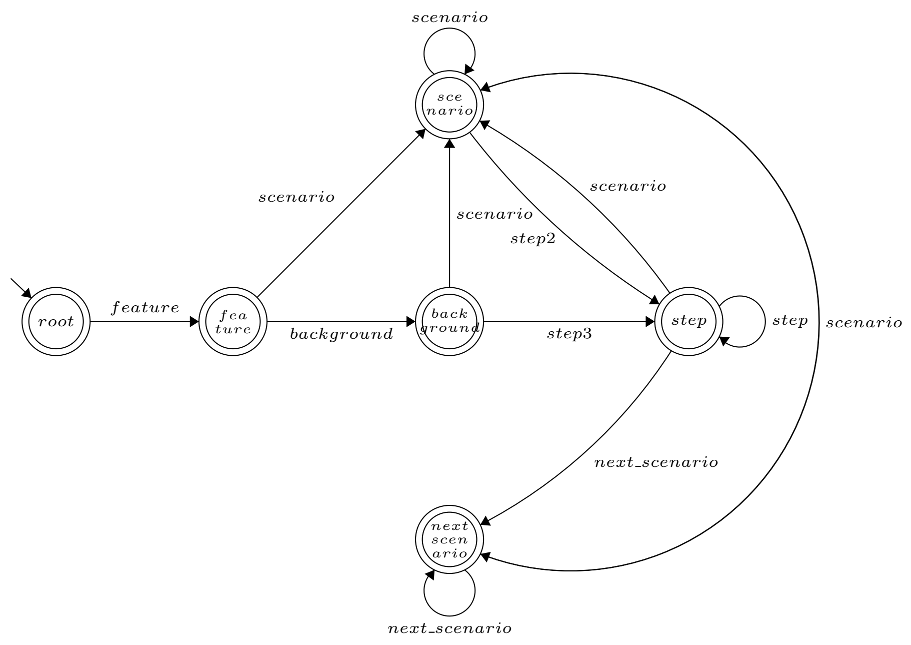


Figure 4.2: The FSM built by Ragel has now a new state and new transitions.

In this way Gherkin* gives Cucumber* a new way to generate scenarios that are not directly written by testers.

4.2 Cucumber*

Thanks to Gherkin*, we are now able to parse feature files which contain also the new keyword. The next problem we face is how we can extend Cucumber in order to archive the following goals:

1. **To execute scenarios written in feature files**. The results of the execution of those scenarios must be the same both using Cucumber* and the original gem.
2. **To generate scenarios which are not written in feature files**. Cucumber* should be able to automatically generate new scenarios and execute them in the same way as the written scenarios.

The flow that Cucumber* follows in order to generate new scenarios can be divided into three main steps:

1. **Graph generation**: Cucumber* creates a graph where each node is a test case. Edges between two test cases exist only if a *Next_Scenario* definition is written in the first node and contains the name of the second one.
2. **Test case paths detection**: every reachable node in the graph is traversed in order to obtain a set of test case paths.
3. **Scenario generation**: A new scenario is generated for each path in the graph.

The focus of the next sections is to explain the techniques used by Cucumber* to generate and execute test case paths.

4.2.1 Graph generation

To generate new test cases, we developed Cucumber*, which builds a graph for each feature file where every scenario is a node and every *Next_Scenario* definition is an edge.

The productions of the grammar allow the parser to recognize lines that contain keywords of the language and at the same time to run semantic actions for those words. More precisely, when the parser is parsing the keyword *Scenario*, Cucumber* uses the existing semantic action for that keyword to add a node to the graph.

Since the number of scenarios in a feature file may be high, we need to choose how scenarios can be represent as nodes in a graph. The first natural choice is to use a progressive number strategy that associates each scenario to a number. However, in this way some information about scenarios is lost and we have no way to recognize a scenario from a number. To solve this problem we label nodes with the names of the scenarios. In this way, if testers use a short and unique name for each scenario in a feature file, we can searching for a scenario in the graph by looking for its name.

To generate edges we write a semantic action for the new keyword. In this way, when the word *Next_Scenario* is recognized, we are able to define the starting and the ending nodes of edges as following: while the starting one is the last node added, the ending one is a node labelled by the plain-text that follows the word *Next_Scenario.*

A problem we face is that the ending node may not exist yet in the graph. For instance consider Figure 4.1: when the *Next_Scenario* word is parsed, the graph contains only the node labelled by "Login". We solve this problem considering that now we are able to recognize if a scenario is already stored in the graph by looking for a node labelled by its name. Hence, two situation must be considered:

1. The graph contains the destination node, so Cucumber* simply adds an edge between those two nodes.

2. The graph does not contain the destination node, so Cucumber* creates a new one. Note that Cucumber* knows how to label that node because the parser sent an event to the Cucumber*'s listener with both keyword and plain-text values. After that, Cucumber* adds and edge between the parent node and the last created one.

The solution we designed to solve this problem leads us to create an oriented graph by assuming that all the scenarios which are the destination of *Next_Scenario* definitions must be defined in the same file. This is a little limitation of our prototype. However, we are planning to extend this vision by allowing testers to define links between scenarios written in different feature files.

The algorithm for building the graph starting from a feature file is shown below.

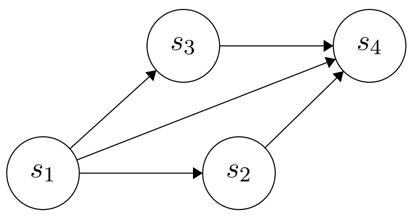
```
Input: A feature file $F$
Output: A graph $G$
Let $G \leftarrow \emptyset$;
Let $S$ be the set of all scenarios in $F$;
foreach $scenario$ in $S$ do
    $G$.add_node($scenario$);
    Let $D$ be the set of all the $Next\_Scenario$ definitions in $scenario$;
    foreach $d$ in $D$ do
        Let $next\_scenario$ be the name of the scenario which appears
        after the Next_Scenario keyword;
        if $next\_scenario$ is not already in $G$ then
            $G$.add_node($next\_scenario$);
        end
        $G$.add_edge($scenario$, $next\_scenario$);
    end
end
return $G$;
```

Algorithm 1: Algorithm for building the graph of test cases.

Note that, if the feature file given in input to the algorithm does not contain *Next_Scenario* definitions, the corresponding graph will not have edges, and no one test case path will be found during the test case paths detection phase. In other words, if the file is written in the Gherkin language, Cucumber* will produce the same output of Cucumber. Figure 4.3 shows an example of how Cucumber* builds a graph starting from a feature file.

```
Feature: Example of Feature

  Scenario: s1
    Next_Scenario: s2
    Next_Scenario: s3
    Next_Scenario: s4

  Scenario: s2
    Next_Scenario: s4

  Scenario: s3
    Next_Scenario: s4

  Scenario: s4
```

(a) An extended Feature file.


$$\begin{tikzcd} & s_3 \arrow[rr] & & s_4 \\ s_1 \arrow[ur] \arrow[urrr] \arrow[rr] & & s_2 \arrow[ur] & \end{tikzcd}$$

(b) The corresponding graph.

Figure 4.3: An example of graph generated by Cucumber*.

4.2.2 Test case paths detection

Once the parsing process of a feature file ends, the final graph is made up by nodes (scenarios) and edges (links between two scenarios). To understand how many scenarios can be generated, we need to know how many test paths there are in the graph. A test case path in a graph is a sequence of test cases (scenarios) that are executed visiting one path. Figure 4.4 shows an example of a test case path.

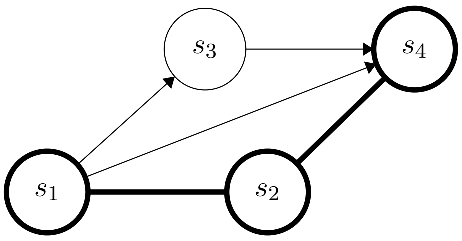


Figure 4.4: An example of a test case path.

The problem of finding test cases paths starting from a graph is known as a graph coverage problem. It can be solved by choosing the best *test coverage criterion* that fits our needs. There are a several of criteria we can consider for our work: Node Coverage (NC), Edge Coverage (EC), Edge-Pair Coverage (EPC), Simple Round Trip Coverage (SRTC), Complete Round Trip Coverage (CRTC), Prime Path Coverage (PPC) [13, p. 25-42]. These criteria can be implemented extending common algorithm such as the Breadth-first search (BFS) or the Depth-first search (DFS). Furthermore, each criterion defines test requirements (TR) in terms of properties of test paths in a graph. For instance, a typical one is met by visiting a particular node or edge.

Our TR is to cover all the possible test paths in the graph generated in the previous step. The only criterion that meets our requirement is the Complete Path Coverage (CPC). However, this method is not feasible for graphs with cycles [13, p. 36]. In fact, graph with loops has an infinite number of paths, so our TR can not be satisfied.

In order to archive the goal of obtaining test case paths from the graph we use a Breadth-first search visit, assuming that the graph in input has no cycles. This is another limit we post to our prototype. However, it is necessary to solve the problem of the infeasible paths in the early phase of the development. Moreover, we plan to use the Prime Path Coverage (PPC, [13, p. 36]) criterion in the near future for finding prime test paths. According to [13], the advantages are:

1. The process is guaranteed to terminate because the length of the longest possible prime path is the number of nodes.

2. It can be computed by a simple dynamic programming algorithm.
3. The algorithm requires loops to be executed.

The Breath-First search algorithm we implemented is shown below. We assume that there is a *.last_path()* method which gets the last subset added in the *curr* set, and a *.last_node()* method which obtains the last node added in a set of sub paths.

```
Input: A graph $G$
Input: An initial node $n$
Output: A list of paths $P$
Mark $n$ as visited;
Let $Q$ a queue;
Let $P$ be the set of final test case paths in $G$;
$Q$.push([$n$]);
$P \leftarrow \emptyset$;
while $Q$ is not empty do
    Let $curr \leftarrow Q$.pop();
    Mark last node of $curr$.last_path() as visited;
    if $curr$.last_path().adj() is empty then
        $P$.add_subset($curr$);
    end
    foreach $v \in curr$.last_path().adj() do
        Let $tmp\_path \leftarrow curr$;
        $tmp\_path$.insert($v$);
        if $v$ is not visited then
            $Q$.push($tmp\_path$);
        else
            $P$.add_subset($tmp\_path$);
        end
    end
end
return $P$;
```

Algorithm 2: Algorithm for obtaining a list of test paths from a graph.

Assuming the graph in Figure 4.3b and the initial node s_1 as inputs to the algorithm above, the result set of test case paths is the following:

$$P = \{ \{ s_1, s_3, s_4 \}, \{ s_1, s_4 \}, \{ s_1, s_2, s_4 \} \}$$

4.2.3 Scenario generation

To generate new test cases we use the set of unique paths P as input. More precisely, each path will be considered as a new scenario that will be executed by Cucumber*. The crucial part of this process is to understand how new scenarios can be created for each path in P.

To understand how we can define a scenario that is made up by other scenarios, we consider every test case as a set of steps. For instance if we consider the scenarios in the subset $\{ s_1, s_4 \}$, we can define the following sets of steps:

$$s_1 = \{ \mathit{Step}_i, \ldots, \mathit{Step}_n \}$$
$$s_4 = \{ \mathit{Step}_k, \ldots, \mathit{Step}_m \}$$

The idea is to create a new scenario which is composed by the union of all the steps included in s_1 and s_4. As result, we obtain one single set, maintaining the order between subsets.

$$\begin{aligned} \mathit{Scenario} &= \{ \mathit{Step}_i, \ldots, \mathit{Step}_n \} \cup \{ \mathit{Step}_k, \ldots, \mathit{Step}_m \} \\ &= \{ \mathit{Step}_i, \ldots, \mathit{Step}_n, \mathit{Step}_k, \ldots, \mathit{Step}_m \} \end{aligned}$$

Now that we know how to compose scenarios in a single one, we have to understand how the scenario defined above can be executed by Cucumber*. The solution to this problem can be found considering the AST explained in Section 2.5. To allow Cucumber* to execute new scenarios, we need to add them as children of the root node *Feature*. Moreover, the new nodes must keep the structure defined by the Table 2.3. Hence, we need a keyword and a plain-text description for each composition.

The keyword problem is solved because we are adding new scenarios to the tree, so we can simply set their keywords to *Scenario*. On the other side, their plain-text values are more difficult to find because each path crossed by the algorithm should have a meaningful name. The reason is that, testers should be able to recognize instantaneously which test case path is failed. A solution we found is to concatenate the names of the single scenarios cross the path using the symbol "→" as follows:

$$\mathit{scenario}_1 \rightarrow \mathit{scenario}_2 \rightarrow \cdots \rightarrow \mathit{scenarios}_n$$

In our case the plain-text values will result the following:

$$\mathit{Names} = \{ \{ s_1 \rightarrow s_3 \rightarrow s_4 \}, \{ s_1 \rightarrow s_4 \}, \{ s_1 \rightarrow s_2 \rightarrow s_4 \} \}$$

Once Cucumber* has the names of the new scenarios, a new node for each one is added to the AST. However, the nodes have no children, so no steps are executed for those scenarios. To add all the steps as children of the new scenarios, it is necessary to find each step node in the AST. The

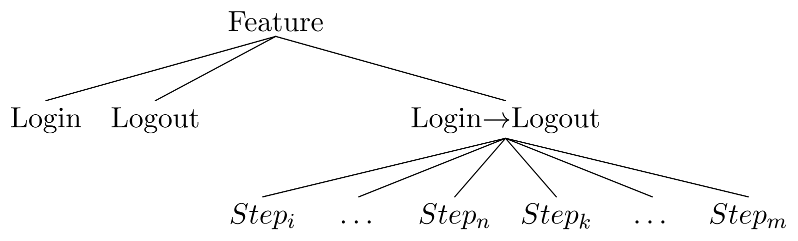


Figure 4.5: An example of an Abstract Syntax Tree built by Cucumber*.

tree is visited and all the nodes found are cloned and added as children of the right scenario. The result of the final AST can be visible in Figure 4.5.

The algorithm which adds new scenarios to the AST is written above. The input are the set of paths produced by the previous step and the AST resulted from the parsing of a feature file.

```
Input: A set of paths $P$, An AST $T$
Output: An AST $T$ which contains also generated scenarios
Let $r$ be the root of $T$;
foreach $p \in P$ do
    Let $name$ be the name of the path;
    $T$.add_node_to_root("Scenario", $name$);
    foreach $node \in p$ do
        Let $S$ be the set of steps in $node$;
        foreach $step \in S$ do
            $step\_node \leftarrow T$.find_node($step$);
            $t$.add_child($step\_node$);
        end
    end
end
return $T$;
```

Algorithm 3: Algorithm for generating new scenarios.

Finally, when the AST is complete and every test case path is added to the tree, Cucumber* executes all the scenarios in the tree by visiting it. No other operations are needed and Cucumber* shows on the output all the scenarios that belong to the tree. An example output of Cucumber* can be seen in Figure 4.6.

As can be seen, no *Next_Scenario* definitions appear in the output. The number of executed scenarios is equal to:

$$\text{Number of written scenarios} + |P|$$

```
Feature: Login and logout as collector

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"

  Scenario: Logout
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page

  Scenario: Login->Logout
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"
    Given I am logged in as "collector@example.com"
    When I sign out
    Then I should be redirected to the login page

3 scenarios (3 passed)
12 steps (12 passed)
0m0.011s
```

Figure 4.6: The result of the execution of Cucumber*.

4.3 Results

Figure 4.7 shows how the feature file shown in Figure 3.1 can be rewritten using Gherkin* and Cucumber*.

```
Feature: Example of Collectors actions

  Scenario: Login
    Given I am on the login page
    When I sign in as "collector@example.com"
    Then I should see "Available Offers"
  Next_Scenario: One donation
  Next_Scenario: Two donations
  Next_Scenario: Logout

  Scenario: One donation
    When I publish a new donation
    Then I should see the last donation in the list of donations
  Next_Scenario: Two donations
  Next_Scenario: Logout

  Scenario: Two donations
    When I publish two new donations
    Then I should see the last 2 donations in the list of donations
  Next_Scenario: Logout

  Scenario: Logout
    When I sign out
  Next_Scenario: Redirect

  Scenario: Redirect
    Then I should be redirected to the login page
```

Figure 4.7: How a feature file can be written using Gherkin*.

We note that two scenarios have been removed because we can execute the same actions by compose scenarios that are written in this file. Moreover, to avoid step duplication, we wrote a new scenario to test the redirect to the login page. All the written scenarios focus on steps that belong to the second category. As result, no duplicate steps occur.

In this feature file there are five *Scenario* and seven *Next_Scenario* definitions. Hence, the result graph shown in Figure 4.8 has five nodes and seven edges.

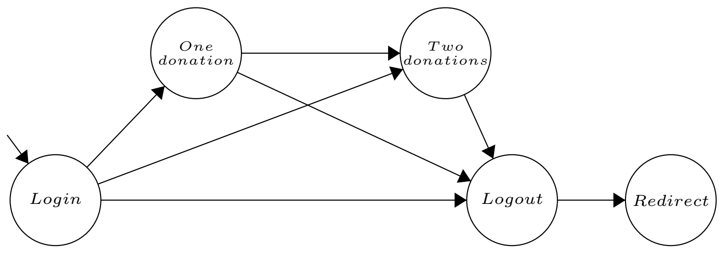


Figure 4.8: The graph built by Cucumber*.

As it can be seen, in the graph there are four test case paths, so the number of scenarios executed by Cucumber* is:

$$\begin{aligned}\text{Total} &= \text{Number of written scenarios} + |P| \\ &= 5 + 4 = 9\end{aligned}$$

The result output of the execution of Cucumber* for the file shown in Figure 4.7 is shown in Figure 4.9. Consider that in the original file the number of steps in the output was equal to 20.

```
9 scenarios (9 passed)
37 steps (37 passed)
0m0.063s
```

Figure 4.9: The result of the execution of Cucumber*.

Conclusion

This thesis describes *Gherkin** and *Cucumber** as extension of two Ruby gems. We started by introducing the BDD technique in Ruby on Rails projects and then we analysed the Gherkin and Cucumber gems. Afterwards we emphasized its benefits and its limitations and then we exposed our tools.

The proposed solution aims to be a novel approach to testing Ruby on Rails web applications following the BDD methodology. The tools developed allow testers to write scenarios and to link them creating a graph of test cases. In this way, not only written scenarios are executed but also a generation process of new test cases is made possible by traversing all the test case paths in the graph.

At the moment, these gems are available with all the limitations described and can be downloaded by testers using the URLs provided. We released both gems under the MIT Licence, so any kind of contribution is encouraged and welcomed. I think that our solution gives testers a fast way to write test cases and I hope it will evolve in the near future with the help of the Cucumber project's community.

The released version of Gherkin* and Cucumber* are far to be complete. We plan to extend Gherkin* and Cucumber* in such a way that testers can use them for all Ruby on Rails web applications. Some of the future work include:

1. **An analysis of the performance of Cucumber***. We want to experiment to how much our approach reduces the time spent by testers to write scenarios.

2. **Find a way to avoid multiple Given-When-Then sentences in the output**. Testers and users may have difficult to read the output of composed scenarios due to complex and long sentences. A solution could be to show only the name of the generated scenarios. In case of errors, Cucumber* may show only the failed step.

3. **Overcome the limit of loops in the generated graph**. The Complete Path Coverage may not be necessary because loops might need to be executed a finite number of times to get an adequate coverage. Hence, a new keyword could be introduced in the Gherkin* language in

order to allow testers to limit the number of times that a scenario can be executed. For instance, testers could write the keyword **Limit_to:** followed by a number that limits the executions of a scenario up to that number. In this way, we can use a graph traversal algorithm to obtain a finite number of test case paths in the graph.

Bibliography

[1] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] D. North. Behavior modification: The evolution of behavior-driven development. *Better Software*, 8(3), 2006.

[3] SkyGate Team. Why should you use behavior driven development (bdd) approach in early stage startups? *Skygate*, April 2014.

[4] Yuan-Hsin Tung, Shian-Shyong Tseng, Tsung-Ju Lee, and Jui-Feng Weng. A novel approach to automatic test case generation for web applications. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 399–404, July 2010.

[5] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005.

[6] Marston Myron. In defense of 100% test coverage. *Dev Blog*, 2012.

[7] Adolfo Villafiorita. *Introduction to Software Project Management.* Auerbach Publications, March 2014.

[8] David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends.* Pragmatic Bookshelf, 1st edition, 2010.

[9] Treehouse Blog. An introduction to rspec. `http://blog.teamtreehouse.com/an-introduction-to-rspec`, 2011.

[10] Matt Wynne and Aslak Hellesoy. *The cucumber book : behaviour-driven development for testers and developers.* Pragmatic Bookshelf, 2012.

[11] Chao Zhang, Xiaoying Bai, Junlong Li, and Renwei Zhang. Automated test case generation for embedded software using extended interface automata. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 292–298, July 2013.

[12] M. Diepenbeck, M. Soeken, D. Grobe, and R. Drechsler. *Towards automatic scenario generation from coverage information.* IEEE, May 2013.

[13] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1st edition, 2008.