# Analysis of Domain Specific Languages for GUI testing: RSpec and Cucumber for Sikuli

Ivan Evgrafov, dilcom3107@gmail.com
Roman S. Samarev, samarev@acm.org
Elena V. Smirnova, evsmirnova@bmstu.ru
Bauman Moscow State Technical University, Russia

Raimund Hocke
Head Developer of SikuliX,
rhocke@me.com, Nidderau, Germany

***Abstract*** **– This article is devoted to new usage of the domain specific languages (DSL) for software with a Graphical User Interfaces (GUI) testing. This paper's authors are a developers of the enhanced software Sikulix, the product which targeted for the programs with graphical interface testing. They propose it as a functional basis for DSL. The SikuliX is considering as an environment for domain specific languages (DSL and DSEL). The results of two specialized testing DSL's estimation are being presented in this paper: the frameworks Rspec which belongs to a Domain Specific Embedded Language (DSEL) group and Cucumber which belongs to a DSL group accordingly. An estimation results of research are shown concerning to the implementation of GUI's image fragments into a application's testing code being under the testing. The scientific novelty of the work is a DSL's integration with graphic as a part of testing scenario. Such approach allows to simplify tasks for experts who create tests as well as for subject experts who are not a programmer but also are able to check a propriety of the testing results. So new approach target specifically at program testing with use of images, we integrated a graphic component into the DSL to create mapping objects which are going to be tested.**

***Keywords – DSL; Ruby; Sikuli; SikuliX; Cucumber; RSpec; image fragment implantation; testing scenario; automation***

## I. Introduction

The practice of using the Domain Specific Languages (DSL) is not new[11, 12]. According to [11] it is good practice to create a special DSL instead of using a General Purpose Programming Language (GPPL) in case of long term software projects. Many computer languages are Domain Specific rather than General Purpose Language. As authors [11, 12] wrote the Domain-specific languages (DSLs) are also called application-oriented, special purpose, specialized, task-specific or application languages. So-called fourth-generation languages (4GLs) are usually DSLs for database applications. Little languages are small DSLs that do not include many features found in General Purpose Programming languages (GPLs). The testers use the GPPL in case if there is no time to create special DSL - in case of the short term software project or if they could not find a proper DSL, or if they could not create their own DSL. And they use a domain-specific language (DSL), a computer language specialized to a particular application domain in such a case [17]. This is in contrast to a general-purpose language (GPPL), which is broadly applicable across domains, and lacks specialized features for a particular domain.

First of all we need to discuss the similarity and difference between the DSL and GPPL. Sometimes it is difficult to see a sharp edge between DSL and GPPL for a number of applications. It is obvious that such languages as Lex, Yacc, $\TeX$, SQL, IDL, HTML are DSLs also. From the other side the originally created as a DSLs Perl, Python, Ruby languages are a GPPLs now. Software testing is a complicated problem. Different software products require different approaches at the different stages of its development and testing. Moreover the involved experts could be using different techniques and instruments including with different languages for testing. Usually the developers use same programming languages for the unit tests as well as for main program. From the other side some of the GUI tests could be executed by separate experts according to customer‘s software requirements.

Let us consider the following approaches to the GUI testing with the aim to find a proper approach which gives a possibility to be controlled by GUI elements:

- same programming means in use for the application and test coding, such approach leads to GUI elements control dependence of the development means and the operational systems. Some tools (Abbot and TestNG) implement this approach to test Java applications written with Swing library [15];
- an involvement of the operating system specific events and commands independently of language being in use, e.g. Ruby language may call methods from Windows API [6];
- with help of some software to record user‘s actions as a script and than to execute the recorded script. The

languages Rational Functional Tester, Open HMI Tester, Froglogic Squish etc. do such a way. These scripts may use the operating system specific events and commands or some properties of programming libraries;

- using some special tools to test special domains like web-applications. This approach is realized by Selenium[1] - testing framework that allows both user's actions recording with Firefox and tests writing in Ruby, Java, Python etc;
- a visual approach to search and automation use image recognition to detect graphical controls and simulating mouse and keyboard to perform actions like a human. This approach is used e.g. in Sikuli [20] and SikuliX[3]. The last one is being under the development of this paper's authors.

We make the following contribution in this paper:

- we discuss about Domain Specific Languages from the point of view applicability and base for DSL development
- we consider four cases GUI testing and propose SikuliX IDE as a basic tool for GUI testing using DSLs
- 'RSpec' examination consider the usage of RSpec testing framework
- 'Cucumber' examination consider the usage of Cucumber testing framework

Finaly we review related work, discuss limitations of our approach and conclude.

In this work the software SikuliX 1.1.0 provides a basis for the GUI tests creation and the Ruby programming language support was integrated into it. So the usage of RSpec and Cucumber DSLs became available in the development environment to access directly. Taking into account that the Sikuli can work with any graphic interfaces the results of the proposed work could be used both for desktop testing as well as for web-application testing. The problem of web applications interfaces testing is important because of statistic research taken from [8].

All tests which authors worked on have been demonstrated with use of LibreOffice Writer 4.

## II. Domain Specific Languages

Most of the testing software tools mentioned above use either GPPL or a special scripting language. There is no strict definition of what is a DSL nowadays, but the main purpose of this article authors is to provide suitable programming abstractions in terms of the domain for end-users [11, 12, 18]. The development of a DSL for some application areas increases starting stage costs but it is compensated later because of simpler code takes less time for domain experts understanding, writing and debugging it. Moreover there are suggestions to realize a DSL-based software design [21].

The area of software testing processes includes both low level decisions like Lava Grammar [16], DSL based on Java annotation techniques like TestNG over Abbot library for Swing-based Java application testing [15] and high level testing tools like DEAL tool [4] that use a GUI for language definition using all names, labels, switch values from forms and dialogues. This article will prove that the RSpec scenario suits better for the low level description whereas the Cucumber scenario - suits for a little bit more abstract actions, however the Sikuli limitations do not allow to go upper till the level of manipulating the window as a stand-alone entity without the proper control element pointment inside of this window.

We have to discuss a difference between DSL and DSEL (Domain Specific Embedded Language) [11]. The DSEL is being realized mainly as a subset of GPPL. And it allows to use all possibilities of GPPL and advantages of the limited syntax of DSL for an application area. It may be used both by programmers who use GPPL and domain experts who use a limited syntax. An example of DSEL is a framework RSpec [5]. The DSL realizes the limited by an application area syntax only. An example of the DSL is a framework Cucumber [9].

DSLs in dynamic languages like Ruby are typically developed as embedded DSLs that can use the existing tooling and platforms of the dynamic language.

The most convenient GPPL for DSEL development is Ruby programming language[13]. The main reasons why authors choose Ruby are flexible syntax and the intention to make program's readability as closer as possible to a natural language (e.g. English). That is, the Ruby approach for specifying DSLs follows the DSL approach itself. Another reason is that the Ruby language is the base for the testing frameworks RSpec and Cucumber. Traditional usage of these tools is testing of web-applications based on the framework 'Ruby on Rails'. But actually both tools are universal and may be used e.g. for testing mobile applications with a touch screen [10].

DSLs can be defined for technical users, such as software developers, testers or architects, as well as for non-technical users, such as business analysts, managers, and so on.

## III. GUI testing

The GUI applications testing has the following specific features. These features are connected to an above mentioned approaches, so we will consider few cases.

1. We know the internal structure of the GUI and we are able to get full access to control elements. This is used in SeleniumHQ [1] where this tool is able to control any supported browser via the page ob-

ject model. Note: there is a special type of web-application testing aimed to check the layout of pages in different browsers.

2. This case is similar to the previous case: we have an application written as local desktop application using Qt, Java-Swing, MFC, .Net or other library. In this case we need to know the specifics of the library used for GUI setup and rendering. This scenario is very complicate for tool development. In many cases the IDE's have appropriate tools to support the developer with testing or there are commercial tools available.

3. In third case we are able to control operating system events targeted at graphical elements [6]. It is possible to simulate any action but without any semantics. Though this approach is rather simple, the test development is very complicate. Moreover there is an obvious dependence from the concrete operating system.

4. If the aforementioned approaches (see sec. Introduction) are not appropriate and if we are not able to control the GUI via a tool or system specific functions, but we are able to simulate user actions via mouse and keyboard, then so called visual approach might fit us. This features are implemented in Sikuli [20] and SikuliX [3]. Both use image recognition options supported by the OpenCV library. It has a possibility to identify GUI components visually by the pixel contents of rectangular areas on the screen and use Java AWT Robot to issue mouse and keyboard actions.

The last case is not simple to realize but it is most universal for a lot of applications being tested.

Before starting to setup tests we should specify general elements of the GUI and answer can they be accessed or manipulated. We should identify elements which have some common features and which control could be automatical. These elements should be generalized with help of utilities making some kind of grouping or templating. Most of GUI applications use a limited number of such standard elements. Non standard GUI elements are rather rare and they require a deeper analysis for their behavior and usage. Classification of elements is demonstrated in [19]. The tables I and II show us a minimal GUI element classification based on the cited works. Table 1 connect GUI elements (column 1) with a category of functions (column 2) and possible user actions which this element reflects (column 3). The interconnection between the user actions and its meaning is shown at the table 2, where the number of action fill in column 1 and its meaning - column 2.

It should be mentioned that the GUI elements in the same category have similar behaviour and may be tested by similar methods of user's control simulation.

The specific part of the application's GUI testing is an interaction with elements that trigger features and functions of the tested application directly. These elements may accept input and/or provide output information. Typical elements are buttons, check boxes, edit fields, comboboxes, lists and menus. Though the proper reaction on these elements actions is different for different applications, they have many common behaviour features such as showing tips to the user or changing the interface after clicks. Such actions and the expected results may be generalized by programmer. There are some special elements like a map, that allows to interact by sliding, or like a real graphical elements like shapes of LibreOffice Draw, that reveal the interactive features to the user. The process of such actions and/or behaviour sets automatical generalizing sets might get too complicate, so special elements may would stay with an individual approach.

The elements of the container category testing is not needed usually in case if the application relies on the elements provided by some system libraries, the last in turn are already have been tested on their normalized behaviour. But an application's behaviour acting with these elements should be tested for sure: e.g. when opening/closing/resizing windows or scrolling it's own content.

Despite of the window's appearance testing is important for users, it is not an aim of this article. The current implementation of SikuliX does not allow to detect skewed or differently rendered images or even images, that appear when they are resized with fewer or additional pixels. But using the right approach or test workflow, detecting the GUI defacing should be possible in most cases.

The process of different GUI elements testing may be similar. Let us consider an example. An application has two buttons. First one opens a window A. Second one opens a window B. Both buttons react on the user's action the same way and the reaction results are similar too. It is obvious how to generalize such elements testing. The only requirement which should be is a failure reporting meaningful for a user and not for a programmer.

In the next section we will demonstrate two approach: the RSpec usage based on classification and Cucumber usage for functional tests as a sequence of simple steps. Our goal is to compare how these two approaches use GUI testing and evaluate cases and to answer which one of DSEL RSpec and DSL Cucumber might be more effective than the other. The following examples are written in the SikuliX IDE with Ruby language support using appropriate RSpec and Cucumber environment templates [2].

## IV. RSpec examination

Now we consider the usage of RSpec [5]. The RSpec offers a DSEL based on Ruby language to get more readable tests than tests written in plain Ruby, by means of an expectation technique and a convenient form of textual descriptions of examples and groups.

Table I: GUI elements in typical applications connected to its user actions codes

| GUI element | Category | Possible user actions codes (see meanings at table 2 ) |
|---|---|---|
| Window | Container | 1-10 |
| Window header | Settings | 7,8,9 |
| Dialog window | Container | 1,2,7-10 |
| Toolbar | Container | 7, 9, 10, 11 |
| Button | Functional | 7, 12 |
| Edit field | Input | 7, 8, 12, 13, 14, 23, 28 |
| Combobox with edit | Input | 7, 12-17, 27, 28 |
| Combobox | Container | 7, 12, 15, 16, 17, 27, 28 |
| List item | Functional | 7, 12, 28 |
| Label | Output | — |
| Table | Container | 7, 15, 18, 19, 20, 27 |
| Cell of table | Input/Output | 7, 8, 9, 12, 13, 14, 23, 28 |
| Checkbox | Input | 24, 25, 28 |
| Radio | Input | 24, 28 |
| Tab switch | Container | 15, 27 |
| Tab | Container | 7, 9, 28 |
| Popup menu | Container | 7, 12, 15, 28 |
| Menu item | Functional | 7, 28 |
| Status bar | Output | — |
| Progress bar | Output | — |
| Document | Input/Output | 7, 8, 9, 12, 13, 14, 23, 27 |
| Text selection | Input/Output | 7, 9, 14, 21, 22, 23 |
| Image in the text | Input/Output | 3, 7-10, 21, 22, 23, 28 |
| Panel splitter | Settings | 26 |
| Slider | Input | 26 |
| Scroller | Settings | 7, 26, 27 |
| Map | Functional | 3, 7-10, 27 |

Table II: Meaning of user action codes

| Act. | Meaning | Act. | Meaning | Act. | Meaning |
|---|---|---|---|---|---|
| 1. | Open | 11. | Dock window | 20. | Change column width |
| 2. | Close | 12. | Hover | | |
| 3. | Change size | 13. | Select text | 21. | Copy |
| 4. | Expand | 14. | Type string | 22. | Cut |
| 5. | Collapse | 15. | Select element from the list | 23. | Paste |
| 6. | Minimize window | 16. | Expand list | 24. | Check |
| 7. | Click | 17. | Collapse list | 25. | Uncheck |
| 8. | Double click | 18. | Click on the cation of row or column | 26. | Slide |
| 9. | Right click | | | 27. | Scroll |
| 10. | Move | 19. | Change row height | 28. | Select element |

The few simple GUI elements testing that activate a visible change on the screen is realized like a function in a traditional programming language, where arguments are images. The RSpec allows to define a common behaviour for several groups using shared_examples_for(group_name) method. E.g. we want to define a group "button". the elements of group "button" should respond to click and optionally to hover events. So for such element testing we need to know how it looks, what will happen when it would be clicked and optionally what would happens when it is hovered. All similar elements in the test are described as it_behaves_like( "button", image_of_element, click_effect) if element doesn't respond to hover and it_behaves_like( "button", image_of_element, click_effect, hover_effect) otherwise. See figure 1.

Another buttons that control the type face of text in the LibreOffice Writer may be described similarly. But some of them require a preparation like e.g. text selection. Therefore we have to select text before. RSpec allows to use special before/after hooks to perform some preparation before each "it"-clause executes in the current context. Moreover marking "it" blocks with metadata, one can choose before which of the scenarios which hooks should be run. See figure 2.

The elements that implement a feature group of the application are usually being placed in containers. These containers should be opened before. In the RSpec this may be realized also with use before/after hooks in shared_context parts. See figure 3. This example demonstrates opening the window before a text manipulation and afterwards confirming the input. Using include_context here, we add hooks declared in shared_context into the current context.

The RSpec is a DSEL and it makes tests much closer to natural English. But at the same time some methods like **shared_context**, **it**, **describe** use Ruby syntax. So the RSpec uses the powerful Ruby language features to describe testing scenarios.

## V. Cucumber examination

The Cucumber is a typical DSL which is aimed to non programmers [9, 7]. The main feature of the Cucumber is a test scenarios describing with use of the Gherkin language, the last one implements a limited subset of natural language. Hence these scenarios are clear both for programmer and for domain expert. Tests written with Cucumber are divided into a pure DSL part that contains sequence of steps written in Gherkin language and a non DLS part in Ruby that describes these steps.

The Cucumber needs several files and the SikuliX IDE can work currently only with one file, because of this we have prepared a special templates for SikuliX that allows to use the Cucumber in the SikuliX IDE. An example of using Gherkin in SikuliX is presented in figure 4.

In this example **Given**, **And**, **When** are special keywords used to mark the beginning of a step. The rest of the line after these keywords is processed by a regular expression in Ruby. See figure 5.

All keywords are the same for any scenarios. Each step definition may be used in any scenario. Therefore this approach is equal to testing templates. But these templates are differ than the templates in RSpec. The main accent in RSpec is made on templates of groups, but Cucumber steps contain templates of scenarios. Cucumber allows to reuse already described steps for the description of new steps. This approach allows to reduce long sequences of steps into one step, when such sequence should be run repeatedly. This means, that we need to do programming in GPPS at the early stages of the test development. And it is possible later to use already described phrases in natural language from the library of step definitions only.

The Cucumber language has before/after hooks and special tags also. The tags are analogous to metadata in RSpec but they are a little bit more powerful due to their ability to perform binary logic operations. The Cucumber allows so called 'Scenario Outlines' construction which reduces several scenarios into one. Note the following example contains a table with samples of images. See figure 6.

Moreover with help of Cucumber it is possible to write scenario descriptions in another languages than English. There are dictionaries of keywords in other languages. Keeping in mind that Ruby works with UTF-8, it is possible to process phrases on any language available in UTF-8. To do this we have to prepare special step definitions for every selected language. The report generated during execution inherits the language from scenario. An example of a Russian language sequence test could be seen at the figure 7.

```
1  # Describe template
2  shared_examples_for "button" do |pattern, click_pattern, hover_pattern|
3    it "should respond to click" do
4      click(pattern)
5      expect(wait(click_pattern, 2)).to_not raise_exception
6    end
7
8    if hover_pattern:
9      it "should respond to hover" do
10       hover(pattern)
11       expect(wait(hover_pattern, 2)).to_not raise_exception
12     end
13   end
14 end
15
16 # Use template
17 describe "background color button" do
18   it_behaves_like "button", [image], [image], Цвет фона
19 end
20
```

Figure 1: An example of 'button' elements checking

```
1  # Describe context
2  shared_context "text selection" do
3    before :each, :text_select => :yes do
4      click(&&&&)
5      dragDrop(&&&&, @@@@)
6    end
7  end
8
9  # Use context
10 describe "bold text style button", :text_select => :yes do
11   include_context "text selection"
12   it_behaves_like "button", [image], Sample text. Sample text., Полужирный (Ctrl+B)
13 end
14
```

Figure 2: An example of shared examples with preliminary actions

## VI. Conclusion

So we have used two testing frameworks: the Cucumber and the RSpec with the SikuliX IDE as a basis for GUI testing. Now it is possible to make a conclusion on the pros and cons about the general approaches.

```
1  # Describe container
2  shared_context "clickable container" do |pattern|
3    before :each do
4      click(pattern)
5    end
6  end
7
8  # Use container
9  describe "text color selection" do
10   include_context "text selection"
11   include_context "clickable container", [image]
12   describe "white color button" do
13     it_behaves_like "button", [image], [image]
14   end
15 end
16
```

Figure 3: An example of container activation

```
1  Feature: Change text properties
2    As a user
3    I want to change properties of selected text
4    In order to make my text less or more readable
5    # Background works as before hook
6    Background:
7      Given text fragment selected
8      And menu opened by click on [Формат]
9
10   Scenario: user chooses some text case
11     Given submenu opened by hover [Регистр]
12     When I click on [Как в предложениях]
13     Then I should see [This text is apper case by default.
                          This text is lowercase by-default.
                          This should be Inverted.]
14
```

Figure 4: An example of Cucumber usage in the SikuliX

There are the RSpec advantages below:

1. The embedded specifications (**describe** keyword) allow to realize tree-like structures having before/after hooks for GUI containers. It is also possible to generate tree-like reports after execution.
2. The test definition asserts in a form of limited natural English.
3. There is a possibility to realize templates for GUI elements with a similar behaviour. It allows to minimize such elements test code.

```
1  # Transform is used here to construct Pattern object from string
2  CAPTURE_PATTERN = Transform /("[0-9]+\.png"|Pattern.*)/ do |pattern|
3    eval pattern
4  end
5
6  # return value of Transform may be used as a part of a regular expression
7  Given /click on (\#{CAPTURE_PATTERN})$/ do |img|
8    wait(img, 2)
9    click(img)
10 end
11
12 Given /opened by hover (\#{CAPTURE_PATTERN})$/ do |img|
13   hover(img)
14 end
15
16 Given /^I should see (\#{CAPTURE_PATTERN})$/ do |img|
17   expect(wait(img,2)).to_not raise_exception
18 end
19
```

Figure 5: An example of step definition for Cucumber

```
1  # Scenario outline description
2  Scenario Outline: user chooses garniture, style and size by clicking
3    Given window opened by click on [Символы...]
4    And tab opened by click on [Шрифт]
5    And <element> is found by scrolling <area header>
6    When I click on <element>
7    And confirm window by click on [OK]
8    Then I should see <effect img>
9  Examples:
10   |element              |area header | effect img                         |
11   | [Adobe Devanagari]   | [Гарнитура] | [perform any actions]             |
12   | [Mangal]             | [Гарнитура] | [ThIS shOULD bE InvERTed.]        |
13   | [Полужирный курсив]  | [Стиль]     | [THIS TEXT IS APPER CASE BY DEFAULT.] |
14   | [22]                 | [Кегль]     | [this text]                       |
15
```

Figure 6: An example of Scenario Outline in the Cucumber

There are the Cucumber advantages below:

1. The test scenarios are written in a nearly natural language.
2. Another languages have been supported by scenario description including Russian and German.
3. There is a convenient technique for processing a sim-

```
1  # language: ru
2
3  Функция: Изменение параметров текста
4
5    Контекст:
6      Допустим выделен фрагмент текста
7      И меню открыто кликом на [Формат]
8
9    Сценарий: Выбрать для текста раскладку "как в предложении"
10     Допустим открыто подменю наведением на [Регистр]
11     Если я кликну на [Как в предложениях]
12     То я должен видеть [This text is apper case by default.
                           This text is lowercase by-default.
                           This should be Inverted.]
13
```

Figure 7: An example of Scenario for the Cucumber in Russian

ilar GUI elements in the same container.

4. The tags are available and the logical operations on them are possible also.

Both RSpec and Cucumber support different forms and format of testing reports (html, xml, json), both may be used with continuous integration system.

So RSpec is more appropriate for pure programmers and English speaking experts. And Ruby code is visible at the primary level.

The Cucumber language allows to read and write testing scenarios and to generate report in many natural languages. Hence the DSL part of Cucumber (Gherkin language) is more appropriate for pure domain experts and customer representatives. But Cucumber is more restrictive about code generalization than RSpec due to containing all scenarios in one feature file inside one context. Therefore you need to mark scenarios with the same tags if you want to use some specific hooks. It is possible to add the background section without tags. A background is much like a scenario in RSpec containing a number of steps, but it runs before all other scenarios. A disadvantage of Cucumber is a lack of hierarchical contexts, which means, that we have to describe tabs and controls for each one separately. The Cucumber is convenient for step definitions, but the usage of it for GUI components description (like separate buttons, checkboxes, radios etc.) is rather complicated than easy.

## VII. Future work

The first one of the several problems connected to described approach is a technical problem. The SikuliX IDE allows to execute any Ruby scripts, but most of testing frameworks require special forms of execution. Examples:

- A framework like 'Cucumber' requires a directory structure with appropriate feature– and step–definition files. Currently it is only possible to emulate this structure. Full support in the SikuliX IDE is not available yet.
- Frameworks 'RSpec', 'Given/When/Then' require a direct access to a Ruby script and the execution via special commands. But the SikuliX IDE controls the execution of Ruby scripts for own purposes, which means that some features are not usable currently.

So it is possible to use the SikuliX as an API already now, but the support for testing frameworks in the IDE needs some specific customizations (e.g. the above mentioned experimental templates) and not all features are fully usable in all cases. The full support for testing frameworks would need specific modifications and additions for each testing framework.

Both RSpec and Cucumber are powerful testing frameworks based on DSEL/DSL. It is possible to realize a library of templates for them. There is an example of modification done for the testing library SeleniumHQ [1] with Capybara [14] project that realizes a DSEL offering more human readable forms of SeleniumHQ functions. The Capybara may also be used with RSpec and Cucumber.

This article authors are going continue to develop the Sikulix software, an enhanced specification for future work is ready, and authors plan to release new version of SikuliX as a basis to support the developement and the

execution of tests using RSpec and Cucumber - SikuliX version 2 in 2015.

References

[1] SeleniumHQ Browser Automation. http://seleniumhq.org/, 2008. [Online; accessed 23-July-2014].

[2] Examples of SikuliX using with testing frameworks. https://github.com/rssdev10/sikulix-ide-templates, 2014. [Online; accessed 23-July-2014].

[3] SikuliX powered by RaiMan. http://sikulix.com/, 2014. [Online; accessed 23-July-2014].

[4] Michaela Bacíková, Jaroslav Porubän, and Dominik Lakatos. Defining domain language of graphical user interfaces. In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *SLATE*, volume 29 of *OASICS*, pages 187–202. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[5] D. Chelimsky, D. Astels, B. Helmkamp, Z. Dennis, D. North, and A. Hellesoy. *The Rspec Book: Behaviour Driven Development With Rspec, Cucumber, and Friends*. Pragmatic Bookshelf Series. Pragmatic Programmers, LLC, 2010.

[6] I. Dees. *Scripted GUI Testing with Ruby*. Pragmatic Bookshelf Series. Pragmatic Programmers, LLC, 2008.

[7] Ian Dees, Matt Wynne, and Aslak Hellesoy. *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. Pragmatic Programmers. Pragmatic Bookshelf, 2013.

[8] Vahid Garousi, Ali Mesbah, Aysu Betin-Can, and Shabnam Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374 – 1396, 2013.

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

[10] Marc Hesenius, Tobias Griebe, and Volker Gruhn. Towards a behavior-oriented specification and testing language for multimodal applications. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 117–122, New York, NY, USA, 2014. ACM.

[11] Paul Hudak. Domain specific languages. In *Chapter 3 in Handbook of Programming Languages, Vol. III: Little Languages and Tools*. MacMillan, Indianapolis, 1998.

[12] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[13] Russ Olsen. Building a DSL in Ruby - Part I. http://jroller.com/rolsen/entry/building_a_dsl_in_ruby, 2006. [Online; accessed 23-July-2014].

[14] M. Robbins. *Application Testing with Capybara*. Community experience distilled. Packt Publishing, 2013.

[15] Alex Ruiz and Yvonne Wang Price. Test-driven gui development with testng and abbot. *IEEE Software*, 24(3):51–57, 2007.

[16] Emin Gün Sirer and Brian N Bershad. Using production grammars in software testing. In *ACM SIGPLAN Notices*, volume 35, pages 1–13. ACM, 1999.

[17] Diomidis Spinellis. Notable design patterns for domain specific languages.

[18] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw. Pract. Exper.*, 38(10):1073–1103, August 2008.

[19] Xuebing Yang. *Graphic User Interface Modelling and Testing Automation*. PhD thesis, School of Engineering and Science Victoria University, Melbourne, Australia, May 2011.

[20] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: using gui screenshots for search and automation. In *UIST*, pages 183–192, 2009.

[21] Uwe Zdun. A {DSL} toolkit for deferring architectural decisions in dsl-based software design. *Information and Software Technology*, 52(7):733 – 748, 2010.