An environment for automatic tests generation from use case specifications

Carolina D. Cunha[1] **and Mark A. J. Song**[1]
[1]Computer Science Department, PUC MINAS, Belo Horizonte, Minas Gerais, Brasil

Abstract— *This paper proposes an environment for the specification of use cases (UCs) and their derivation in automated functional tests for web applications. The proposed environment is based on our own domain specific language (DSL), formalized by means of a BNF grammar (for the Xtext framework), offering all the amenities and usability of the Eclipse IDE. The proposal seeks to tighten the relation between system specifications and tests, aiming to facilitate and incentivize the use of automated tests and their continuation, since they are frequently abandoned due to cost and deadline limitations. All of the tools used in the process, from specification to generation, execution and management of tests, are free and in the public domain.*

Keywords: functional-tests, use-cases, automation

1. Introduction

Tests are fundamental activities in software development, because they are the last product evaluation resource before delivery to end users. Among the various types of tests, functional tests, in which systems are evaluated concerning their inputs and outputs, without considering their internal workings, stand out. Because they are based on system's specifications, functional tests are largely used, because software products must behave like what was defined with users during requirement analysis [1].

The demand for faster development of software products while retaining quality calls for the very frequent execution of tests, especially regression tests, to assure that new functionality did not compromise the behavior of previously implemented and validated requirements [2]. Although regression testing is important, it is one of the most costly stages of software development and maintenance. According to [3], software tests can account for 30 to 50% of total development cost, and according to [4] it is estimated that regression testing is responsible for up to 80% of total testing costs, and up to 50% of total software maintenance costs.

The realization of automated tests, in addition to reducing time spent testing the software, allows for an indirect increase in software quality, since efforts can be focused in other types of tests or in tests that cannot be automated [5]. Automating repetitive tasks not only reduces costs, but also improves precision, since humans are slow and prone to error when dealing with such tasks.

It is perceivable that the generation of functional test cases is still a predominantly manual and arduous task [6], as is their execution, according to experience obtained in large companies - an especially complicated scenario in large-scale and long-lasting projects [5].

For test automation, test scripts are written that replicate manual tests. The generation of functional test scripts can be done by manual coding or by a recording tool (like the Selenium IDE [7]), using the "Capture and Replay" technique.

At first sight, the technique seems like a feasible solution, because it requires no programming skills. However, if a test needs to be altered, so does the recorded script - which would require such skills - or a new recording must be manually made [5]. Besides, scripts generated by those tools are linear, with hard-coded inputs and comparisons, with no possibility of reusing a script to compose new tests, not to mention being hard to write and maintain. Furthermore, the link between original and generated code is weak, making it difficult to pinpoint specific reasons for an error identified during testing.

In spite of the advantages in using automated code, they are considerably costly to maintain, because for every alteration in the system under test (SUT), tests must be reviewed and updated as needed. If they fail to be updated due to cost or deadline reasons, they become worthless. According to [8], only a small part (sometimes less than 20%) of tests are automated because of these difficulties.

One way to reduce testing efforts while still guaranteeing effectiveness would be to automatically generate Test Cases (TC) from artifacts that were previously used during development [9]. One of the most widely used ways to capture software requirements is through Use Case (UC) description, a central mechanism recommended in Unified Process (UP) and in many modern methods [10], [11]. TCs derived from UCs can be used to check if system requirements (UCs) were correctly implemented [3], [9].

The goal of this work is to generate test scenarios from UCs and, from these scenarios, generate executable TCs that are converted into automated test scripts that retain a connection to the specification.

2. Related Work

Many studies seek to use UCs to generate TCs or test scenarios. In [9], test scenarios with sequencing based on

UC's textual pre and post conditions are generated and described in controlled language. Generated scenarios have a high abstraction level, opposite to what is proposed here, a more concrete-level approach.

In [10], textual UCs described in a controlled language and a domain model are used as inputs. Data inconsistencies, like entities being described with multiple names, UC operations that do not refer to a concept operation in the domain model, among others. Scenarios are generated by UC composition, and their execution is simulated in a state machine with a tool called UCEd. Like in [10], we propose a domain model with possible value information and intend to generate scenarios with UC sequencing, but for real test execution.

In [12] and [13], a UC specification language named SilabReq, based on Xtext [14], is presented. Domain model, system operation list, UML UC model and state, sequence and activity diagrams are generated from UCs. [13] proposes dividing the specification into different abstraction levels, since UCs are used by people of different roles, with different needs, during software development, ranging from end users, requirement engineers, to designers, developers and testers. Like [12] and [13], we propose a UC specification language, with varied abstraction levels, using Xtext as base. However, since our focus is generating executable tests, the proposal encompasses elements beyond what is explored in SilabReq.

According to [15], to contemplate the needs of every role involved in a software development project, it is necessary to specify UCs in many notations. A notation set is thus proposed, based on structured text, and interaction and activity diagrams, with defined rules to convert one into the other. A test-friendly notation is mentioned as an important and integral part of the set, but was not mentioned in the paper, apparently because it was not worked on by the group.

In [16], a tool called TestGen is used to generate activity diagrams and possible scenarios from UCs. A correctness study is conducted for the generated TCs using mutant analysis. Our proposal also uses the aforementioned tool, but distinguishes itself by generating executable test scripts.

In [17], UCs are specified in an environment called Text2Test, based on the Xtext framework, in which validations vary according to user profile. We also propose an Xtext-based environment with validations, but focusing on test execution.

[2] presents Webspec, a visual language with diagrams, used to capture browsing, interaction and interface characteristics for web applications. Webpsec was built as an Eclipse plugin and uses mockups (UI doodles) for simulations. It enables the generation of code and Selenium tests. Like [2], our solution generates executable tests for Selenium and uses Eclipse, but is based in a descriptive language instead of diagrams.

Finally, in [18], an approach to generate test scenarios from UCs with contracts formalized in OCL and sequence diagrams. The required formalization in [18] elevates the technical level necessary to understand and maintain UCs.

3. Proposed Approach

For the present work, we developed a DSL for UC description, formalized in a BNF grammar (fig 1). The grammar, fed to the Xtext framework, is used in the generation of the language's concrete syntax (by means of a meta-model), for the proposed environment.

We opted to use the "Fully Dressed" use case template, presented in [19], which has a positive response in practical experiments reported by the author. This template uses one text column and numbered steps with a convention of numbers and letters for alternate steps. This numbering structure supports identification and generation of scenarios.

UCs can be described with varying levels of detail, according to the project's stages. In [11], three detailing levels for writing UCs are presented, and it is suggested that UCs be written in an essential style especially in the beginning of specification - when the focus must be on intention, and not on UI - and in a concrete/detailed style in the following stages, when the UI project is available.

In this paper it is proposed that every step of the UC in the abstract level be followed by its detailing, where UI details are inserted. By adding UI information to UC steps, test scripts can be generated. The use of an environment where UCs are kept linked to tests helps reflecting into the latter changes made in the former. Even if the changes are not entirely automatically realized, they become motivated by the connection of UC steps to their detailing and by validation-generated warnings. Besides, UCs produce useful debugging information when a test fail (fig 8). So, as can be seen in figure 1, UCSteps are composed of Steps, that can be of types: Action (User interactions with the screen), Verification (Assertions that validate system responses) or Storage (Storing values for later use).

In addition to UCs, UIs must also be described - with their elements and menus - optionally with a domain model (data dictionary), that helps to resolve ambiguity, registering composite terms and the relationship between the elements. Changes to the dictionary or UI generate warnings to users if they break references in UC steps (or vice-versa) (fig 5). Fields and messages are referenced in the use cases through internal IDs. When changes occur, they only need to be reflected in the UI specification, which facilitates maintenance (PageObject pattern [20]).

When indicating an input value to be inserted into a UI element, fixed values can be used directly, or rules for dynamic generating input in execution time can be stipulated. Dynamic values can stem from field type description (type, minimum and maximum values), from an expression or from a list of values. The rule specification for dynamic value generation can be made in a UC step, in a determined

```
UseCase:
 'UC' name=ID '{'
 ('context' descriptionUC=STRING)?
 ('keywords' keywords=STRING)?
 'actor' actorUC=[Actor]
 'precondition' precondition=PreCondition
 'postcondition' postcondition=STRING
 'mainscenario' mainscenario=MainScenario
 'extensions' extensions=Extensions
 '}';
MainScenario:
   (UCSteps+=UCStep)(',' UCSteps+=UCStep)*;
UCStep:
 seq=INT description=STRING
 '{'   (steps+=Step)(',' steps+=Step)*  '}';
Step:
 Action | Verification | StoreValue;
Action:
 Access | Inform | ExecuteUC | DealWithPopup |
 Click | WaitUntilCondition;
Access:
 'Access' ('menu' menuId=[Menu] | 'link' linkId=[Link]);
Inform:
  'Inform' field=[Field] 'with' value=Value ;
Value:
 (simpleValue=STRING | dynamicValue=DynamicValue |
 ref=DictionaryElementProperty);
```

```
ExecuteUC:
 'Execute' exec=[UseCase] (parms=Parms)?
('with extension' '=' extension=STRING)?;
DynamicValue:
 DataTypeAndSize | ValueList | MinMaxLimit | Expression
;
GlobalMessages:
 'GlobalMessages' '{' msgs+=Message (',' msgs+=Message)*
'}'
;
Screen:
 'UI' screenId=ID '{'
 fields+=Field (',' fields+=Field)*
 (messages=ScreenMessages)?
 '}';
Field:
 (obrig='*')? name=ID ':' htmlType=HtmlType
uiIdentification=UIIdentification
('description' '=' fieldDescritpion=STRING)?  (dynamicValue=
DynamicValue)?
;
UIIdentification:
'label' '=' fieldLabel=STRING | 'id' '=' field=STRING
;
HtmlType:
 (text='TEXT' | listbox='LISTBOX' | radio='RADIO' |
checkbox='CHECKBOX' | button='BUTTON') ;
```

Fig. 1: Some rules of proposed DSL .

screen field or a determined property in the data dictionary. If specified in the dictionary, it is possible to vary the specification for ranges, called equivalence classes. Should a rule be used in multiple UCs, it should be defined in a single location (the dictionary or the screen) and be referenced in the UC steps. If this rule is used in various screens, it becomes more interesting be defined in the data dictionary

Test generation must be run every time an alteration to the specification is realized. For test generation, it is first necessary to generate test scenarios, traversing path possibilities in the basic and alternative flows (an example is shown in section 4).

Among the many options to generate the paths of a UC, like a graph or FSM (finite state machine) conversion, we opted to treat the UC like an activity diagram, because it is easily understood by users, helping them visualize and validate the completeness and correctness of UC paths.

3.1 The environment

This paper presents an environment for the specification of UCs - and their unfolding into TCs - based on the open-source framework Xtext. Through this framework, it is possible to obtain a specific editor for a language, on the Eclipse IDE [21], with all the characteristic support, which favors productivity and usability features, like: support to "folding", "autocomplete", "syntax highlighting", "structure outline" and "cross-referencing", in addition to easily managing artifact versions.

The environment validates, in editing time, the informed text, according to the supplied grammar. During data input, the allowed options in each point are presented ("autocomplete" resource). When it is a cross-reference point, options for the referenced type are offered. When specifying a menu access, for example, a list containing all declared menu options is shown. Semantic validations are also considered during editing, and added to the code generated by the framework. An example of such validation is the verification that every mandatory field (indicated by '*') were filled, in a UC's steps, before the step that requires submitting the screen. When an error is found, it is indicated in the text and in a tab that helps fixing it ("Quick Fix" resource) (fig 5). Due to the fact that validations happen in editing time, the author can iteratively refine the UC. Standardization in writing UCs is aided and guided by the environment, improving its quality and facilitating the identification of its components, that can be used for automatic transformations afterwards [12].

For scenarios generation, we opted for the TestGen [16] tool, which takes the high-level UC steps as input and generates test scenarios (.tol files) and an activity diagram (.xmi file) as outputs, for each UC. Test scenarios are generated for every path obtained from the diagram.

Test scripts are generated in Java, and begin their execution using JUnit [22]. The scripts use, for interaction with the UI, the Selenium WebDriver [7] tool, for simulating user navigation. This tool was chosen because it is a free tool and in the public domain.

Test executions are done in a declarative way, using the

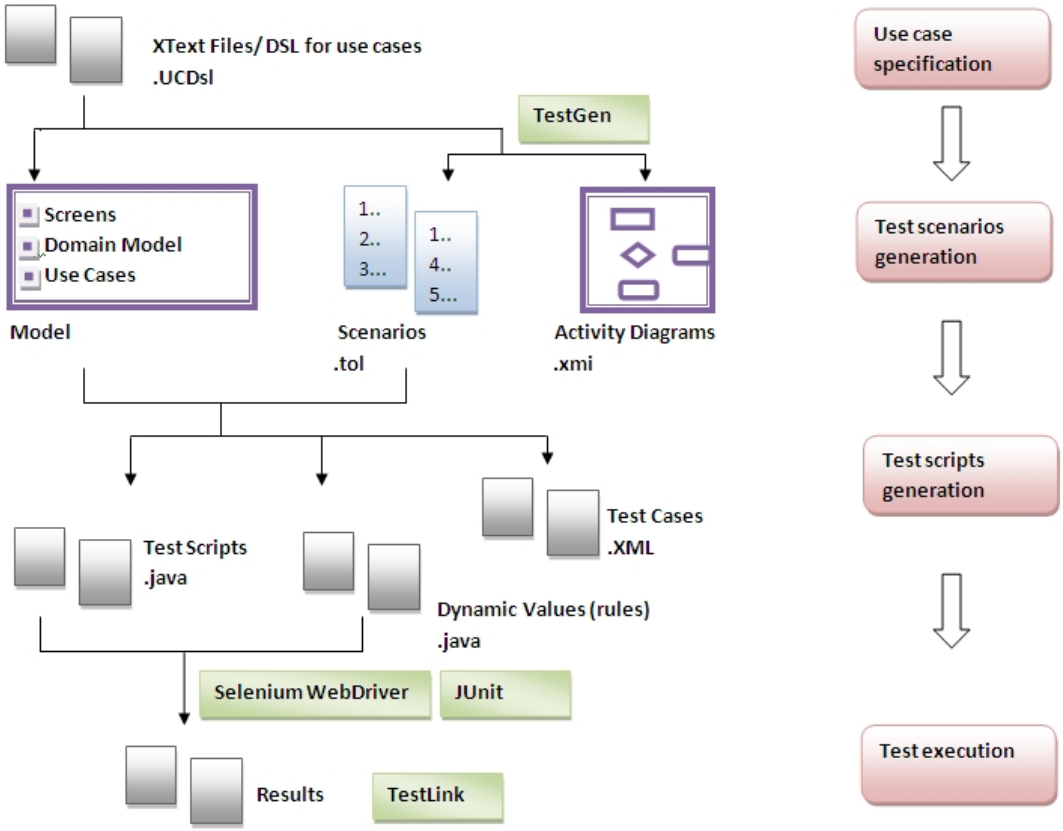

Fig. 2: Proposed environment structure.

UC name. It is also possible to execute all or a set of tests, formed by UCs that correspond to keywords informed in the execution. Keywords related to each UC are previously added to the specification, in addition to UC name. It is possible to test a UC's main scenario or every possible scenario. UC sequencing is allowed, by stating so in the pre-conditions or within steps of the scenario. UCs invoked from other UCs have the main scenario executed, unless extensions are indicated he should go through.

For each execution, dynamic values are generated according to the rule specified for each field.

Results are stored in the TestLink [23] environment, because it is a free and complete tool for test management, that allows users to centralize automated and manual tests, keep an execution history (with evidence logging), and obtain management reports. Each execution is recorded, along with its results. Errors results in the logging of the following evidence: a screenshot and a Java error stack trace, containing information about the UC, scenario and the step in which the error occurred, in addition to the runtime flaw (fig 8). These evidences, with the SUT's logs, help identifying what caused the errors.

Figure 2 shows the structure of the environment proposed and artifacts used in each phase.

4. Experiments

In this section we present a viability study, for the UC "Create Student", presented in figure 3. Data dictionary and user interface specification are also presented (fig 4).

UseCases.UCDsl_01 | Screens.UcDsl_01

```
UC CreateStudent {
  context "Registering a new student"
  keywords ""
  actor Secretary
  precondition ""
  postcondition ""
  mainscenario
    1 "User [access] UI [Create_Student]." {
          Access menu Create_Student
      },
    2 "User [informs] data of new [student]." {
    3 "User [submits] the request " {
    4 "System [shows message] of success." {
  extensions
    3a "Incomplete Data"
      3a 1 "System [shows message] requesting data"{
      3a 2 "System [goes to] step 2"{
    4a "Already registered student"
      4a 1 "System [shows message] error message "{
}
```

Fig. 3: Example of UC: Create Student.

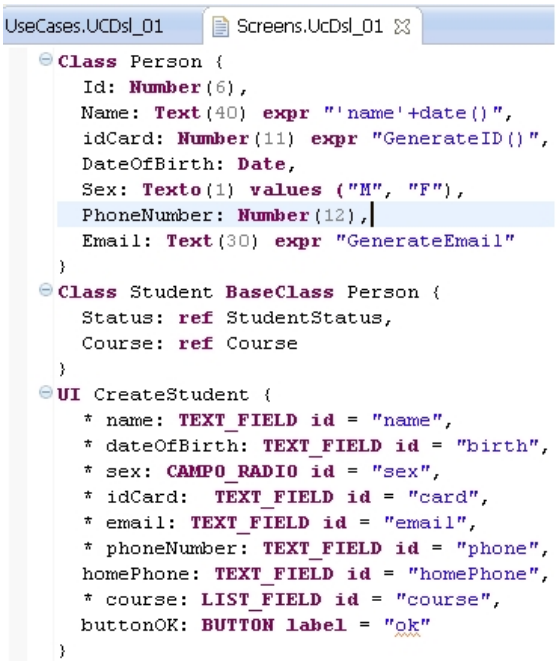

Fig. 4: Example of data dictionary and UI specifications.

Simulating a specification modification motivated by an alteration of the SUT, it was removed, from the IU description, the field "homePhone". An error in the UC that references the removed field is pointed out by the environment (fig 5).

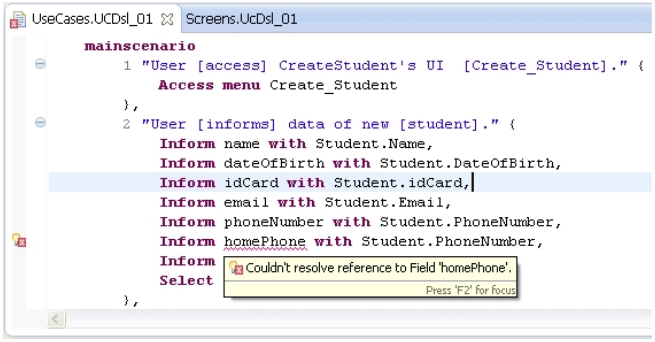

Fig. 5: Error pointed out by the environment.

Scenario generation for the UC CreateStudent resulted in 4 scenarios, as can be seen in figure 6. Test scripts were generated for the scenarios and tests were ran with positive results.

So, to simulate the discovery of an error in a regression test, we inserted a flaw in the SUT, in the routine that reads course data, causing a collateral effect in the routine that adds a student, which conducts some validations regarding courses to allow the addition of a student.

Figure 7 shows the execution results: the first one ended normally and the second one failed with an error. By clicking

<Sequence> 1, 2, !3a, 3, !4a, 4, End, </Sequence>
<Sequence> 1, 2, !3a, 3, 4a, 4a1, End, </Sequence>
<Sequence> 1, 2, 3a, 3a1, 3a2, !3a, 3, !4a, 4, End, </Sequence>
<Sequence> 1, 2, 3a, 3a1, 3a2, 2, !3a, 3, 4a, 4a1, End, </Sequence>

Fig. 6: Scenarios generated for the UC Create Student.

on the UseCaseName_StackTrace evidence, details about the error can be seen (fig 8). As can be noticed, class names that appear on the error stack bring useful information to aid in discovering the reasons behind the error, pointing to which step/scenario/UC the error occurred in. In this case, in step 4, scenario 1_2_3_4 of the CreateStudent UC.

5. Conclusion and Future Work

It was concluded that using the proposed environment was feasible for a real system, according to case study. This study covered the CRUD functionality, and, in a second phase, will treat varied and more complex features. Between the two stages, it was decided to develop some mechanisms to facilitate the specification, as a new kind of step that informs all fields of a screen and submit it, a way to describe variations of a screen by means of extensions, and a mechanism to parameterize instances of similar use cases. Some limitations were found during the experiments, like the non-treatment of asynchronous steps and lack of permission for variations of more than one level, inside the alternate flux, situations that can be seen in UC22, in [19].

As future work, first and foremost, we intend to consider this second level of variation (which involves altering the TestGen tool) and the above mentioned enhancements. Additionally, we intend to: develop a mechanism to automatically capture the screen structure; allow the execution of tests for only UCs that suffered changes, between two versions; allow the execution of tests with "pairwise" technique, dealing with boundary-value analysis and equivalence class partitioning techniques (the element specification format already allows the generation of this variation); deal with screen modeling, like in [2], since the employed technologies are related; generate mechanisms to facilitate importing specifications created with other tools; generate the activity diagram in formats compatible with free UML tools, using XMI as base.

6. Acknowledgements

We would like to thank FAPEMIG for the financial support.

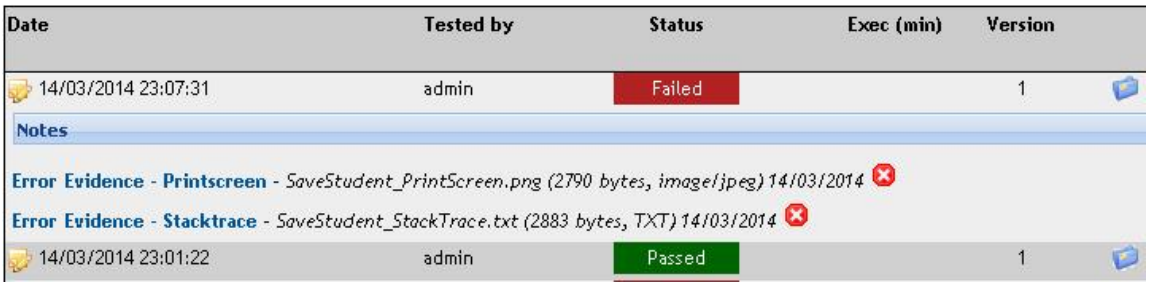

Fig. 7: Execution results.

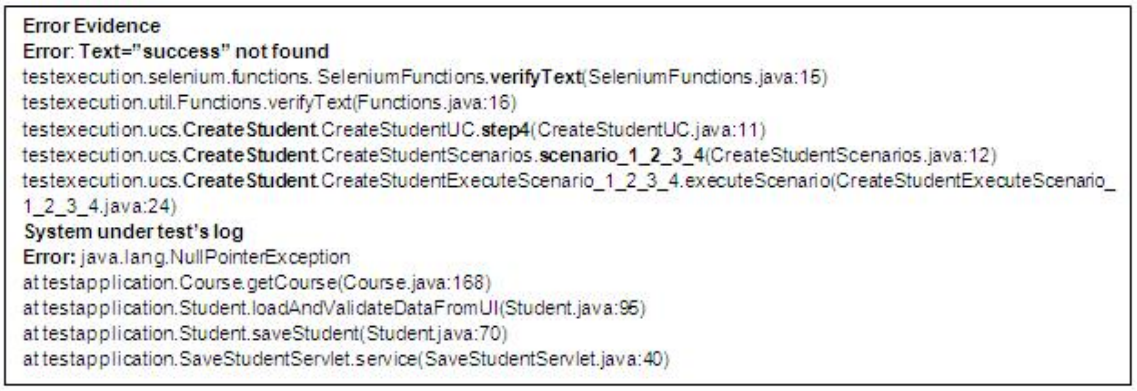

Fig. 8: Error evidence in the execution of the CreateStudent UC.

References

[1] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. Bookman, 2011.

[2] E. R. Luna, G. Rossi, and I. Garrigós, "Webspec: a visual language for specifying interaction and navigation requirements in web applications," *Requirements Engineering*, vol. 16, no. 4, pp. 297–321, 2011.

[3] J. Heumann, "Generating test cases from use cases," *The rational edge*, vol. 6, no. 01, 2001.

[4] M. J. Harrold, "Reduce, reuse, recycle, recover: Techniques for improved regression testing," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009.

[5] P. Pedemonte, J. Mahmud, and T. Lau, "Towards automatic functional test execution," in *Proceedings of the 2012 ACM international conference on Intelligent User Interfaces*. ACM, 2012, pp. 227–236.

[6] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "Autoblacktest: a tool for automatic black-box testing," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1013–1015.

[7] Selenium, "Selenium," http://www.seleniumhq.org, Mar. 2014.

[8] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 881–891.

[9] S. S. Some and X. Cheng, "An approach for supporting system-level test scenarios generation from textual use cases," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 724–729.

[10] S. S. Somé, "Supporting use case based requirements engineering," *Information and Software Technology*, vol. 48, no. 1, pp. 43–58, 2006.

[11] C. Larman, *Applying UML and Patterns An Introduction to object - oriented analysis and design and iterative develop*, 3rd ed. Prentice Hall, 2005.

[12] D. Savic, I. Antovic, S. Vlajic, V. Stanojevic, and M. Milic, "Language for use case specification," in *Software Engineering Workshop (SEW), 2011 34th IEEE*. IEEE, 2011, pp. 19–26.

[13] D. Savic, A. R. da Silva, S. Vlajic, S. Lazarevic, V. Stanojevic, I. Antovic, and M. Milic, "Use case specification at different levels of abstraction," in *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. IEEE, 2012, pp. 187–192.

[14] Xtext, "Xtext," http://www.eclipse.org/Xtext, Mar. 2014.

[15] M. Smialek, "Accommodating informality with necessary precision in use case scenarios." *Journal of Object Technology*, vol. 4, no. 6, pp. 59–67, 2005.

[16] J. J. Gutierrez, M. J. Escalona, M. Mejias, J. Torres, and A. H. Centeno, "A case study for generating test cases from use cases," in *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*. IEEE, 2008, pp. 209–214.

[17] A. Sinha, S. Sutton, and A. Paradkar, "Text2test: Automated inspection of natural language use cases," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 155–164.

[18] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 140–155, 2006.

[19] A. Cockburn, *Writing effective use cases*. Pearson Education, 2001.

[20] PageObject, "Page object," http://martinfowler.com/bliki/PageObject.html/, Mar. 2014.

[21] Eclipse, "Eclipse," http://www.eclipse.org/, Mar. 2014.

[22] Junit, "Junit," http://junit.org/, Mar. 2014.

[23] Testlink, "Testlink," http://testlink.org/, Mar. 2014.