Tool support for Generating User Acceptance Tests

Guy Camilleri, Leandro Antonelli, Pascale Zaraté, Juan Cruz Gardey, Jonathan Martin, Amir Sakka, Diego Torres, Alejandro Fernandez

ICDSST 2020 PROCEEDINGS – ONLINE VERSION
THE EWG-DSS 2020 INTERNATIONAL CONFERENCE ON DECISION SUPPORT SYSTEM TECHNOLOGY
I. Linden, M.T. Escobar, A. Turón, F. Dargam, U. Jayawickrama (editors)
Zaragoza, Spain, 27-29 May 2020

Tool support for Generating User Acceptance Tests

Guy Camilleri[1], Leandro Antonelli[2], Pascale Zarate[3], Juan Cruz Gardey[2], Jonathan Martin[2], Amir Sakka[2,4], Diego Torres[2,5,6] and Alejandro Fernandez[2,6]
[1]SMAC group, IRIT, 118 route de Narbonne, 31062 Toulouse Cedex 9, France
[2]LIFIA, Facultad de Informática, UNLP, Argentina
[3]ADRIA group, IRIT, Université de Toulouse, 2 rue du Doyen Gabriel Marty, 31042 Toulouse Cedex 9, France
[4]IRSTEA Clermont-Ferrand 9 Avenue Blaise Pascal, 63178 Aubiere France
[5]Departamento de Ciencia y Tecnologia, UNQ, Argentina
[6]CICPBA, Buenos Aires, Argentina
{camiller,zarate}@irit.fr,
{lanto, jcgardey, jcgardey, dtorres, casco}@lifia.info.unlp.edu.ar,
amir.sakka@irstea.fr

ABSTRACT

Software testing, in particular acceptance testing, is a very important step in the development process of any application since it represents a way of matching the users' expectations with the finished product´s capabilities. Typically considered as a cumbersome activity, many efforts have been made to alleviate the burden of writing tests by, for instance, trying to generate them automatically. However, testing still remains a largely neglected step. In this paper we propose taking advantage of existing requirement artifacts to semi-automatically generate acceptance tests. This paper extends a previous paper in which we use Scenarios, a requirement artifact used to describe business processes and requirements, and Task/Method models, a modelling approach taken from the Artificial Intelligence field. The proposed approach derives a Task/Method model from Scenario (through rules) and from the Task/Method model specification, all alternatives in the flow of execution are provided. Using the proposed ideas, we show how the semi-automated generation of acceptance tests can be implemented by describing an ongoing development of a proof of concept web application designed to support the full process.

Keywords: User Acceptance Tests, Scenarios, Task/Method model, Agriculture Production

INTRODUCTION

Developing software still remains a very complex process involving several actors and consisting of different steps. The testing step remains as one of the biggest problems, and it is frequently avoided. As a consequence, the resulting system can fail to meet users' expectations, rendering it useless. Our objective is to develop a strategy to make the testing step easier, generating User Acceptance Tests (UAT) in a semi-automatic way from requirements artifacts. Many software development methods use, in the early stages, steps to clarify business processes and specify requirements. These processes are often used to define the UAT. A semi-automatic generation of UAT can with few efforts support the software

engineers to elicit, to clarify and to discuss the business processes and the requirements by showing some implications of their analysis/modeling. Theses analysis can result in new modifications and developments of the model of business processes and requirements. Therefore, a semi-automatic generation of UAT constitutes a decision support for the modelling of business processes and requirements. To do this semi-automatic generation, we combine two modelling approaches: Scenarios, from the requirement engineering field and Task/Method models, from the Artificial Intelligence field, particularly knowledge-based systems [3]. A first work has been done (see [1] and [2]) which proposes to use a wiki website for describing Scenarios, and to translate these Scenarios in Task/Method model in a semi-automatic way.

Figure 1 depicts the overall proposed process. First, the users describe scenarios thanks to a website application after, the translation rules are applied to generate the corresponding task/method model. These steps were already proposed in a previous work [1]. The obtained task/method model is then executed by an execution engine which produces an Execution Tree (ET). A ET is a data structure representing all possible executions of the task/method model (hence, all possible flows of actions and tests). Test cases can be extracted from this ET. In this paper, we will focus on the last two steps: execution engine and the test cases generation.

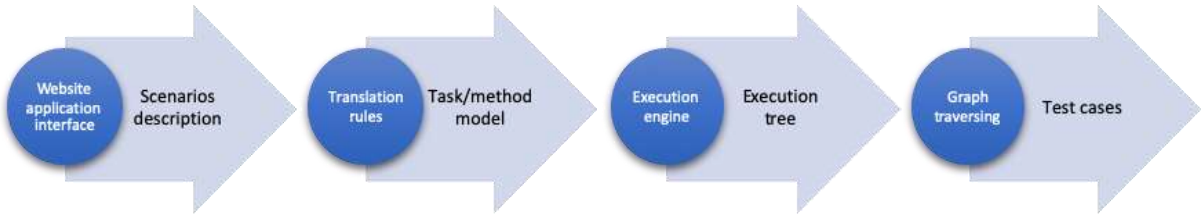

Figure 1: Test cases generation process

This work is applied to the RUC-APS project. RUC-APS is a H2020 RISE-2015 project, aiming at Enhancing and implementing Knowledge based ICT solutions within high Risk and Uncertain Conditions for Agriculture Production Systems. In this context we will use a scenario based on agriculture production. The rest of the paper is organized as follows: we first introduce related work, then present the background introducing scenarios and the Task/method paradigm. In the third part, we describe the two last steps of our approach (see Figure 1) which will be illustrated by a Task/Method model generated from a scenario based on agriculture production. Finally, we show our conclusions and future work.

RELATED WORK

Garousi et al. [4] describe six steps in test cases automations: (i) test-case design, (ii) test scripting, (iii) test execution, (iv) test evaluation, (v) test results reporting and (vi) test management and other test engineering activities. Our approach has the aim of designing test-cases. So, we provide a technique to cope with the first step (test-case design). Takagi et al. [5] describe a strategy to develop a graph that model the histories of test case execution. Although the authors deal with low level histories related to hardware testing, their proposal is similar to our proposal, since we generate a tree with all the different scenarios that need to be tested. Monpratarnchai et al. [6] propose a tool to automatically generate test cases for Java applications. They analyze the source code and derive a script using a symbolic language. After that, Junit code is generated. Our strategy is similar, since we analyze Scenarios, the source description of the requirements and Task / Method model language is used to specify criteria that allow to obtain test cases. Stoyanova et al. [7] propose a

framework for testing web app. The framework has two main parts: (i) test case generation and (ii) test case execution. Although we have to execute Task / Method model script, it is needed to obtain the test cases. That is, the tree that we obtain is the final test cases that is needed to test the application. Chatterjee et al. [8] propose an approach to automatically generate test cases from Use Cases. Bouquet et al. [9] propose a similar although they use class diagram and state machine to derive the tests. They explore all the alternatives in the flow of the dialog as well as the preconditions and they generate all the tests needed. The difference with our approach is that they rely on state while we rely on actions. We consider that every action can be success or fail, why they rely on every state of the different elements included in the situation.

BACKGROUND

Scenarios

Scenarios can be used in different stages of software development, from clarifying business process and describing requirements, to providing the basis of acceptance tests [10]. There is a distinction between application domain (the real world) and the application software (the machine) [11]: during business process modelling and requirements elicitation, Scenarios describe events in the world, while in system specification, they describe events in the machine. Scenarios are stories about people and the activities they perform to reach certain goals, parting from a setting and counting with some resources. Their description ranges from visual (storyboards) to narrative (structured text) [12]. Leite et al. [13] propose a template with six attributes to describe Scenarios in a textual way: (i) Title, it is the name of the scenario to identify it, (ii) Goal, conditions and restrictions to be reached after the execution of the Scenario, (iii) Context, conditions and restrictions that are satisfied and constitute the starting point of the Scenario execution, (iv) Actors are agents that perform actions during the Scenario to traverse the path from the context to reach the goal, (v) Resources, products and elements used by the actors to perform action, and (vi) Episodes: steps executed by the actors using the resources beginning at the context to reach the goal.

The text descriptions in Scenarios follow a fixed structure. In particular, episodes must be written with full sentences describing the subject, the action they perform, and if necessary the resource used. The following example describes partially some Scenario for farmer packing products. The example also includes the cases to consider for testing the scenario. These test cases do not belong to the original structure of the scenario:

Scenario: detect stress in crops of tomatoes and peppers
Resources: Sensors
Actors: System
Episodes:
The sensor reads the temperature
The sensor reads the level of humidity
The sensor reads the intensity of the light
The system determines if it is a stressful condition
Test cases:
If some sensor can not read the data the system do not have the input necessary to infer a prediction.
All the sensors can read the data, but the system does not have historical information to infer a prediction

Scenario: collect information
Resources: Sensors
Actors: System
Episodes:
Several sensors collects information about the temperature
The system calculates the average to determine the temperature
Test cases:
There is a problem collecting the information
There is a problem summarizing the date

Task/Method Paradigm

The task/method paradigm is a knowledge modelling paradigm (mainly from the artificial intelligence field [14], [15]) that sees reasoning as a task. Knowledge is expressed in a declarative way, making it easy to process by execution engines or planners [1]. A task/method model is composed by a ***domain model*** and a ***reasoning model***. The former describes the objects of the world being used (directly or indirectly) by the latter, similarly to an application ontology. It is often described in UML language and implemented with OO languages. The reasoning model describes how a task can be performed. It uses two modelling primitives: a ***task***: is a transition between two world state families (an action) and is defined by the following fields: *Name, Par, Objective* and *Methods*. A ***method*** describes one way of performing a task. A method is characterized by the following fields: *Heading, Prec, Effects, Control* and *Subtask.*

The task's field *Name* specifies the name of the task. The field *Par* contains the list of parameters, that is, all objects handled by the task. For example, in a task *Read*, the parameter list could be *(sensor, temperature)* which are domain objects (from domain model) used by the task *Read*. We will write *Read(sensor, temperature)*. The list of methods which can be applied to perform a task is described in the field *Methods*. A terminal task is a directly executable task (without described methods). The method's field *Prec* contains conditions that must be satisfied to apply the method. The execution order of subtasks is described in the *Control* field, and sub-tasks are recorded in the *Subtask* field. Note that, by essence, Task/Method models are hierarchical. Here we explained only the fields used in this work, see [2] for a full reference.

User Test Cases Generation

In this work, we make the following assumption. We consider that we dispose of a Task/Method model obtained in the two first steps of our approach (Figure 1, see for more details [1]). The execution of tasks in the task/method model can only succeed or fail. Specifically, only the terminal tasks succeed or fail directly, the execution status (success or failure) of the other tasks results only from the status of the terminal tasks. Under this assumption, all possible executions of a task/method model will correspond to the propagation of two possible execution status (success or failure) of terminal tasks. In the previous example (see also Figure 2), the "Read(sensor,temperature)" task has one method with two terminal tasks: "Collect information (system, sensors, temperature, data)" and "Summarize data (system, data)". These terminal tasks can succeed or fail. So if both succeed, the "Read(sensor,temperature)" task succeeds and if one of them fails, then the "Read(sensor,temperature)" task fails. In our approach, we consider that each user test case corresponds to an execution path. In the "Read(sensor,temperature)" example, two user test cases can be extracted from the following execution paths: "Collect information (system, sensors, temperature, data)" fails therefore "Read(sensor,temperature)" fails and, "Summarize data (system, data)" fails therefore "Read(sensor,temperature)" fails.

To generate user test cases, it is possible to generate user test cases directly from the task/method model, or to generate all execution paths and extract user test cases from these execution paths. We have chosen the latter option which is more flexible and separates the execution process from the extraction process. Thus, the execution engine produces all execution paths in the form of Execution Tree (ET). User test cases are extracted from the ET and possibly with some natural language processing tools. An ET contains all possible executions of one task. It is composed of two types of node: the etask nodes which represent the executed tasks and the emethod nodes the executed method. In the figure 2, an ET is

drawn for the task “Detect stress”. Boxes correspond to etasks, ovals to emethods and arrows link etasks to emethods. One task can be executed by several methods, and one method can have several emethods according to the execution status (success or failure) of subtasks. In the figure 2, the etasks and the emethods with gray background are etasks and emethods that failed.

The following algorithm describes the execution engine that produces an ET for one etask. Each etask and each emethod have a boolean attribute “failure” (true for failure and false for success). etasks and emethod are instantiated from Tasks and Methods of the task/method model. By default, the failure status is false for all etasks and all emethods. If an etask *et* is terminal, one new emethod is added with a copy of *et* in which the failure status is true. In this way, for each terminal etask, there exist two versions of this etask, one with the failure status to false and the other with the failure status to true. If an etask is not terminal, all applicable methods are instantiated and executed. A method is executed by launching the code in its control field which will rerun the Execution_engine function on some etasks in the subtasks field.

```
Execution_engine(et:ETask)
 if et is a terminal then
            set false to failure status of et;
            et_failure=Duplicate et with failure status to true;
            em_failure=Duplicate the emethod of et with failure status to true;
            link et_failure and em_failure to the parent task of the emethod of et;
else
            methods= all methods of et;
            for all m in methods do
                        em= instantiate m;
                        link em to et;
                        if em is applicable then
                                    execute control field of em
                        end if
            done
end if
return et;
```

As an ET contains all ways of executing an etask, user test cases can be extracted by traversing the ET from the failed terminal etasks (leaves of ET) to the initial etask (root of ET). The proposed process has been applied to the “detect stress in crops of tomatoes and peppers” scenario described previously. The figure 2 presents the ET obtained by the execution engine tool. For generating UAT, we simply traverse the ET from the leaves which fail to the root. Each extracted branch corresponds to one UAT. In the current implementation, UAT are generated by a direct translation from these ET branches. We obtained the following UAT.

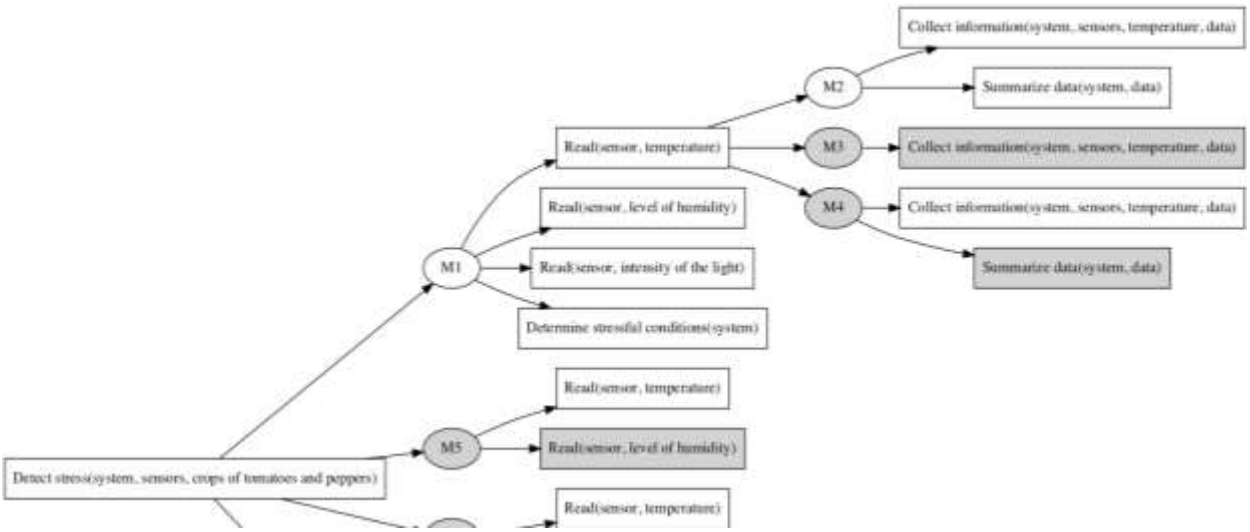

Figure 2. Execution tree for Detect stress task (success white background and failure gray background)

- Detect stress(system, sensors, crops of tomatoes and peppers) fail because Read(sensor, temperature) fail because Collect information(system, sensors, temperature, data) fail.
- Detect stress(system, sensor, crops of tomatoes and peppers) fail because Read(sensor, temperature) fail because Collect information(system, sensors, temperature, data) succeed, but Summarize data(system, data) fail.
- Detect stress(system, sensor, crops of tomatoes and peppers) fail because Read(sensor, temperature) succeed, but Read(sensor, level of humidity) fail.
- Detect stress(system, sensor, crops of tomatoes and peppers) fail because Read(sensor, level of humidity) succeed, Read(sensor, temperature) succeed, but Read(sensor, intensity of the light) fail.
- Detect stress(system, sensor, crops of tomatoes and peppers) fail because Read(sensor, intensity of the light) succeed, Read(sensor, level of humidity) succeed, Read(sensor, temperature) succeed, but Determine stressful conditions(system) fail.

CONCLUSION

In this paper we presented a way to generate UATs from a Task/method model. This work follows previous work ([1] [2]), where users describe scenarios through a web application and from this description, translation rules are applied to generate the corresponding task/method model. Our approach is to use an execution engine that generates an execution tree representing the trace of all possible executions. From this execution tree, UATs can be extracted using graph traversing and natural language processes. In the current version of the execution engine, only textual descriptions of tasks are processed. In future work, we want to study how to use a domain model in the form of object-oriented model in order to integrate UATs related to the domain model in the execution engine.

Acknowledgements

Authors of this publication acknowledge the contribution of the Project 691249, RUC-APS: Enhancing and implementing Knowledge based ICT solutions within high Risk and Uncertain Conditions for Agriculture Production Systems (www.ruc-aps.eu), funded by the European Union under their funding scheme H2020-MSCA-RISE-2015

REFERENCES

1. Leandro Antonelli, Guy Camilleri, Julian Grigera, Mariangeles Hozikian, Cécile Sauvage, “A Modelling Approach to Generating User Acceptance Tests”. 4th International Conference on Decision Support Systems Technologies (ICDSST 2018), May 2018, Heraklion, Greece. ⟨hal-02289948⟩
2. L. Antonelli et al “Wiki Support for Software Use Cases” Special Issue on Promoting Sustainable Decision-making, Kybernetes Journal, ISSN: 0368-492X, Emerald Publishing, Bingley, Reino Unido, accepted March 27, 2019.
3. G. Camilleri, J.-L. Soubie, and J. Zalaket, “TMMT: Tool Supporting Knowledge Modelling,” in *Knowledge-Based Intelligent Information and Engineering Systems*, vol. 2773, 2003, pp. 45–52.
4. V. Garousi and F. Elberzhager, "Test Automation: Not Just for Test Execution," in IEEE Software, vol. 34, no. 2, pp. 90-96, Mar.-Apr. 2017. doi: 10.1109/MS.2017.34
5. T. Takagi and K. Noda, "Partially developed coverability graphs for modeling test case execution histories," 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), Okayama, 2016, pp. 1-2. doi: 10.1109/ICIS.2016.7550886
6. S. Monpratarnchai, S. Fujiwara, A. Katayama and T. Uehara, "An Automated Testing Tool for Java Application Using Symbolic Execution Based Test Case Generation," 2013 20th Asia-Pacific Software Engineering Conference (APSEC), Bangkok, 2013, pp. 93-98. doi: 10.1109/APSEC.2013.121
7. V. Stoyanova, D. Petrova-Antonova and S. Ilieva, "Automation of Test Case Generation and Execution for Testing Web Service Orchestrations," 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering, Redwood City, 2013, pp. 274-279. doi: 10.1109/SOSE.2013.9
8. R. Chatterjee, K. Johari. “A prolific approach for automated generation of test cases from informal requirements”. SIGSOFT Softw. Eng. Notes 35, 5, October 2010, pp 1–11. doi: https://doi.org/10.1145/1838687.1838702
9. F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux. “A test generation solution to automate software testing”. In Proceedings of the 3rd international workshop on Automation of software test (AST ’08). Association for Computing Machinery, New York, NY, USA, 2008, pp 45–48. doi: https://doi.org/10.1145/1370042.1370052
10. I. Alexander and N. Maiden, “Scenarios, Stories, and Use Cases: The Modern Basis for System Development,” *IEEE Comput. Control Eng.*, vol. 15, no. 5, pp. 24–29, 2004.
11. M. Jackson, “The world and the machine,” in *Proceedings of the 17th international conference on Software engineering - ICSE ’95*, 1995, pp. 283–292.
12. R. Young, *The requirements engineering handbook*. 2004.
13. J. Leite and A.P.M. Franco “A strategy for conceptual model acquisition”, In Requirements Engineering conference. IEEE. doi:10.1109/ISRE.1993.324851, pp 243–246.
14. F. Trichet and P. Tchounikine, “DSTM: A framework to operationalise and refine a problem solving method modeled in terms of tasks and methods,” *Expert Syst. Appl.*, vol. 16, no. 2, pp. 105–120, 1999.
15. G. Schreiber, H. Akkermans, A. Anjewierden, R. De Hoog, N. R. Shadbolt, and B. Wielinga, *Knowledge Engineering and Management: The CommonKADS Methodology*, vol. 99. 2000.