AGUTER a platform for Automated Generation of User Acceptance TEsts from Requirements Specifications

Leandro Antonelli, LIFIA–CICPBA, Facultad de Informatica, UNLP, La Plata, Argentina.
Guy Camilleri, IRIT, Universite Toulouse III–Paul Sabatier, Toulouse, France.
Diego Torres, LIFIA–CICPBA, Facultad de Informatica, UNLP, La Plata, Argentina. Also Depto. CyT, Universidad Nacional de Quilmes, Bernal, Argentina.
Pascale Zarate. IRIT, Universite Toulouse I–Capitole, Toulouse, France.

Abstract

[Purpose] This article proposes a strategy to make the testing step easier, generating User Acceptance Tests (UATs) in an automatic way from requirements artifacts. [Design/methodology/approach] This strategy is based on two modeling frameworks: Scenarios and Task/method paradigm. Scenarios is a requirement artifact used to describe business processes and requirements, and Task/Method paradigm is a modeling paradigm coming from the Artificial Intelligence field. The proposed strategy is composed of four steps. In the first step, scenarios are described through a semantic wiki website. Then scenarios are automatically translated into a task/method model (step two). In the third step, the Task/method model obtained in step two is executed in order to produce and store all possible achievements of tasks and thus scenarios. The stored achievements are saved in a data structure called execution tree. Finally, from this execution tree (step four), the user acceptance tests are generated. [Findings] The feasibility of this strategy is shown through a case study coming from the agriculture production systems field. [Originality/value] Generally, test design approaches deal with a small number of variables describing one specific situation where a decision table or workflow is used to design tests. Our proposed approach can deal with many variables because we rely on scenarios that can be composed in order to obtain a tree with all the testing paths that can arise from their description.

Keywords: User Acceptance Tests; Requirements Specifications; Scenarios; Task/Method model; Agriculture Production Systems

1. Introduction

Developing software still remains a very complex process consisting of different steps where several actors from the IT team and stakeholders are involved. For several years, agile methodologies have arisen as a silver bullet in order to cope with this problem. Nevertheless, the testing step remains one of the biggest problems. Some estimations show that more than half of the budget in software development is spent on testing (Ammann and Offutt, 2016). Although it is a good decision in order to provide tools and train the testers, it is observed that most skilled software testers become developers, analysts, or architects (Capretz et al., 2019). As a consequence, the testing stage remains a challenge and the resulting system can fail to meet users' expectations.

Thus, this paper proposes an approach to make the testing step easier, helping testers to design the User Acceptance Tests. The proposed approach provides a good coverage of the situations that should be considered to test and elaborate a systematic and complete set of alternatives that need to be explored in order to test. The approach combines two modeling approaches: Scenarios, from the requirement engineering field and Task/Method models, from the Artificial Intelligence field, particularly knowledge-based systems and semantic support (Camilleri et al., 2003; Antonelli et al., 2019). Scenarios are stories about people and their activities to reach certain goals, starting from a setting and using some resources. Scenarios can be used to describe the application context (as a business case) and can also be used to

describe software functionality (as Use Cases). Thus, our approach can be used in different levels of detail. The Task/Method paradigm proposes to model human reasoning using tasks and methods. In this modeling paradigm, knowledge is expressed in a declarative way, making it easy to process by execution engines or planners.

The proposed approach consists in transforming the Scenarios in the form of Task/Method models using them to automatically generating test cases. Tests generated explore all the alternative and combination of flows in the execution of the Scenarios. Thus, a systematic revision of the condition is assured. Scenarios can be used to describe different levels of abstractions and the test obtained from them can be different types of tests. If the Scenarios describe high level requirements (also known as Business cases), test cases will be User Acceptance tests. This is the commonly use of the Scenarios. Nevertheless, Scenarios can also be used to describe low level requirements (something similar to Uses Cases). In this situation, the tests obtained will be unit tests. Nevertheless, in any case (high or low level), tests derived by the proposed approach can be used to verify and validate the Scenarios written in an early stage, in order to correct the Scenarios and prevent a misunderstanding at the beginning of the project with impact in the whole cycle. In order to contribute to solve this issue specifically, a first work has been done (Antonelli et al., 2018) which proposes to use a semantic wiki website for describing Scenarios, and to translate these Scenarios in Task/Method model in a semi-automatic way.

Generally, tests design is made identifying some variables and creating a decision table with these variables. Our approach used Scenarios as input, so we can consider that every episode is one different variable of traditional approaches using decision table. Thus, every episode (variable) could be right or wrong (success or fail), and our approach designs all possible combinations. Thus, if we apply the strategy proposed to one Scenario, our approach would obtain the same result obtained by creating a decision table with traditional techniques.

Nevertheless, traditional techniques do not scale up when they have to deal with many variables since the combinations follow an exponential function. That is, one variable means two different tests, two variables means four different tests, three variables means eight tests, and so on. Scenarios commonly have more than four episodes. Moreover, it is common to have a composition of Scenarios. That is, one episode of a scenario can be described as another scenario with many episodes. For example, Scenario 1 can have four episodes: episode 1.1, episode 1.2, episode 1.3 and episode 1.4. And episode 1.1 can also be described as a new Scenario 1.1. And this scenario 1.1 will have several episodes. This composition can have many levels and the amount of alternatives can grow significantly. In this situation it is important to identify infeasible paths, that is, combinations that are not possible to happen or are not important to tests.

Thus, the novelty of our approach is building one unique and integrated tree with all combinations needed to test. This tree is performed by our approach automatically analyzing the different scenarios. Thus, the tree can be mapped to the Scenarios that give origin to it and the scenarios can be analyzed to decide if some branch is worth testing or not (because it is an unfeasible path).

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. Section 2 introduces other approaches in the literature that are related to this article. Then, the Scenario description, Task/Method Paradigm, and Test Case description are introduced as background in Section 3. The article approach is detailed in Section 4. Finally, Section 5 concludes the article with conclusions and further work.

2. Related work

Testing is a very complex activity since the key in test design is finding mistakes, errors and problems instead of solving them which is the objective of almost the rest of the steps in the software development life cycle: architecture, design, code, etc... That is why some researchers are concerned about the cogni-

tive processes of software testers in order to contribute to the field (Enoiu et al., 2020). Moreover, certain domains are complex, so testing is even more critical and test design demands much effort. For example, software testing in automated vehicles is crucial to launch safe and reliable vehicles (Masuda, 2017). Several issues in the software testing of automated vehicles have been raised including extremely large space of test input and high cost of test executions. This is specifically our main concern: a large space of test input. Thus, our proposed approach uses Scenarios to analyze their episodes in a combinatorial way to obtain the whole space of alternatives. Our approach can be categorized as specification-based, structured based, and experienced-based according to (ISO/IEC/IEEE, 2014). This is commonly used in automated vehicles and similar complex domains. Moreover, Scenarios are also models and (Ramler and Klammer, 2019) use models to increase the coverage and reduce the effort of test design. They report their experience of applying model-based testing in several real-world projects from industry where they were able to minimize the risks and to reduce the effort in testing. Dos Santos et al. (Dos Santos et al., 2018) present a literature review about test design approaches and they show that there is no approach with tool support that uses text description with natural language, in particular behavior driven technique using scenarios. And these are the key elements of our approach. We state that natural language is very important to obtain description directly from end users, the ones who will use the application, therefore they need to accept it. Moreover, Scenarios describing the behaviour of the domain are a good tool since scenarios tells stories and people are naturally trained in that activity. There are other works to derive UAT from Use Cases (Hsieh et al., 2013) (Bystricky´ and Vranic´, 2017) or acceptance criteria described in the form of Given-When-Then(Pandit et al., 2016). But all require an effort to build these models and Agile methodologies which do not use these artifacts (Jeeva Padmini et al., 2016). will demand and even extra effort.

Svensson et al. (Svensson and Regnell, 2015) state that automating testing is a shared concern in software engineering. Testing a software requires generally a lot of effort form the programmers. They should imagine all the possible errors that could be made by the end-users. They also have to work with a lot of rigour in order to test all possible cases that are included in a code. They can miss some specific cases and it is the reason why they need to be assisted. Garousi et al. (Garousi and Elberzhager, 2017) propose an approach with six steps: (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. Stoyanova et al. (Stoyanova et al., 2013) propose a framework for testing web app with two main parts: (i) test case generation and (ii) test case execution. It is important to remark that both proposal include one first step (separated from the others) related to the design of the test cases. Our proposed approach does not execute the tests, it only deals with the design.

Monpratarnchai et al. (Monpratarnchai et al., 2013) propose an approach to generate test cases described in JUnit from Java source code. It is important to obtain unit tests, but our proposal agrees with Bjarnason et al. (Bjarnason and Borg, 2017) where they obtain test case from requirements. We believe that test cases should be linked to user needs (i.e. requirements) in order to validate that the application satisfies them. Lipka et al. (Lipka et al., 2015) propose a combined approach, because they derive test cases from requirements and source code. They consider narrative requirements enriched with annotations to connect the specification to source code. Khamaiseh et al. (Khamaiseh and Xu, 2017) propose an interesting technique based on Use Cases, in which they determine misuse case to test vulnerabilities. Philip et al. (Philip et al., 2017) approach is similar, since they analyze a model with safety requirements to generate fault trees representing functional hazards. Then, test cases for validation of mitigation of hazards are generated automatically from the model.

There are many proposal that are based on requirements. Some proposal are based on conditions, restrictions or states. These elements could be captured by the requirements using Use Cases, formal languages, states machines or workflows diagram. And these elements are used as the input of the approach. Chatterjee et al. (Chatterjee and Johari, 2010) propose an approach to derive test cases from Use Cases, as well as (Bouquet et al., 2008). Although they analyzes the alternatives in the flow of actions, they also analyze the preconditions stated in the Use Cases, thus they finally rely on a state machine. By contrary, our approach rely on the combination of all the possible actions (and their result). Pandit et al. (Pandit et al., 2016) propose an approach to design User Acceptance Tests (similar to our approach) but they base their proposal on acceptance criteria written in the form of Given-When-Then template. Thus,

they also rely on states that are arranged in a dependency graph. Lei et al. (Lei and Wang, 2016) propose a framework to analyze testing constraints in requirements, that is, another way of considering states. Huaikou et al. (Miao Huaikou and Liu Ling, 2000) analyze Z specifications, in particular, the prior and posterior state of every operation to generate test cases.

There are many other proposal that are related to the steps that are implicitly linked to every requirements, for example, as in (Hussain et al., 2015). That is the essence of our approach. Nevertheless, some distinction should be made. Some proposal uses Use Cases or similar products, where the description of the requirements has a big precision. While some other uses Scenarios, where the description is more related to he business than the application. Our approach is in this last category. Then, there are approaches that analyze the description inside a Use Case or Scenario, while other analyse the relationship between Use Case or Scenarios. Our approach, relies on both things. Our approach analyzes the internal description of the Scenarios, but, they can be also described as another Scenarios that gives an overview of the problem.

Considering Use Cases, there are several approaches. Hsieh et al. (Hsieh et al., 2013) propose an approach to analyse the steps of the Use Cases in order to determine all the alternatives to design tests to cover all the possibilities. Then, other approaches are based on the external relationship of the Use Cases. Lizhe Chen et al. (Lizhe Chen and Qiang Li, 2010) consider the relationships between the Use Cases. Budha et al. (Budha et al., 2011) propose a similar approach based on Use Case diagrams. This approach generates test cases to detect use case dependency faults using multiway trees. They transform the use case diagram into a tree and they traverse the tree. This is similar to our approach since we traverse a tree to obtain all the alternatives. Boucher et al. (Boucher and Mussbacher, 2017) also analyse workflow models (Use Case Maps) to transform them into Acceptance Test Cases that can be automated with the JUnit framework. Nogueira et al. (Nogueira et al., 2014) propose an approach to generating test cases from use cases with specific definition of control flow, input and output. Vieira et al. (Vieira et al., 2006) propose a similar approach using annotated UML Activities Diagrams

Considering Scenarios Entin et al. (Entin et al., 2009) propose an approach focused in obtaining test independent from the platform. That is, they obtain very general User Acceptance Tests as the one obtained by our approach. Takagi et al. (Takagi and Noda, 2016) describe a strategy to develop a graph about the sequence of test case execution related to hardware testing, that is, very detail and specific situations in contrast with the essence of the Scenarios. Hussain et al. (Hussain et al., 2015) also provide an approach to design test considering dependencies between Scenarios. Nomura et al. (Nomura et al., 2014) model the business context in a matrix representing the dependency between business process, then from the perspective of profiles is design the tests to cover the different situations. Sarmiento et al. (Sarmiento et al., 2016) propose a similar approach using scenarios.

3. Background

This section describes the two main modeling techniques used in our proposed approach. These two techniques are not our proposal, but they are used in our approach. The first technique is a template for describing the Scenarios, the input of our approach. The Scenarios capture the behaviour of the domain and the tests are designed regarding this behaviour. The second technique is the Task / Method model, a conceptual model that provides the execution capability of the specification of the Scenarios in order to obtain the execution tree of all the alternative workflows.

3.1. Scenario

Scenarios can be used in different software development stages, from refining business processes to defining requirements giving the support of acceptance tests (Alexander and Maiden, 2004). There is a gap between the context domain (real world) in which the software application will be executed in a computer (Jackson, 1995). Scenarios describe both: real-world events like business processes and re-

quirements in the machine. Scenarios describe the activities that people perform (using resources) to reach a goal. Scenarios need to be linked in order to describe a bigger story. Thus, Scenarios usually include description of conditions that need to be satisfied in order to be carried out. There is a wide granularity in their descriptions from the use of visual artifacts such as storyboards to structured text (Young, 2004). A Scenario described with text and a template with six attributes was proposed by Leite et al. (d. P. Leite and Franco, 1993). It is described in Table 1. The text that describes a Scenario follows a fixed structure. Mainly, the episodes should be written with a full sentence that contains the subject and the predicate (or clause). If it is necessary, the predicate can include some object (direct object). Thus, the sentences has a structure of a semantic triple: subject/predicate/object (also known as RDF triple). In order to combine the text model of the Scenarios and a semantic representation, the tool uses an intermediate model called Language Extended Lexicon (LEL). It is a glossary that is described using text using a structure similar of the RDF.

Table 1
Scenario template

Title: Identify the Scenario by a name.
Goal: It defines the conditions to be reached after the execution of the Scenario.
Context: Also known as pre-conditions that should be satisfied at the beginning of the Scenario execution.
Actors (and agents): Stakeholders that execute actions to reach the goal from the context.
Resources: The elements and products that are manipulated or used by actors to perform actions.
Episodes: Are the steps that actors execute to reach the goal.

The Table 2 describes an example of a Scenario where stress in crops of tomatoes and peppers is detected. This Scenario is an example of two different levels at the same time. This Scenario can be considered as a situation in the domain as well as some requirements for an specific application. That is, a farmer can perform the tasks about sensing certain weather variables and analyze them to identify if the conditions are stressful or not. And, this Scenario can also be fully implemented taking advantage of Internet of the Things. The Scenario describes a situation where there are several sensors in a greenhouse that measure the temperature, humidity and sun radiation. It is common to have different temperatures in different places inside a greenhouse. Thus, after reading the sensors, the average of measures is calculated to estimate a global measure. Then, after some time, the variables become uniform in all the greenhouse. This could be achieved because physics laws or with the help of fans. The tests related with these Scenario are related with sensors that can fail or even the calculus are not correctly performed (Table 3). The Scenario of collecting temperature reading is described in detail in another Scenario (Table 4). The process implies some technological step (sensor reading) and manual steps (records the values and calculate the average). Thus, there could be failures in any one of the steps (Table 5)

Table 2
Scenario Detect stress

Title: Detect stress in crops of tomatoes and peppers.
Goal: Prevent a stressful situation to correct it before damage to crops occur.
Context: Greenhouse.
Actors (and agents): Farmer.
Resources: Thermal sensors, humidity sensors, luminosity sensors.
Episodes:
The farmer collects temperature reading from thermal sensors.
The farmer collects humidity reading from humidity sensors.
The farmer collects sun radiation reading from luminosity sensors.
The farmer determines the possibility of a stressful condition.

3.2. Task/Method Paradigm

A conceptual model is used to represent scenarios in our approach. In conceptual models, the knowledge

is expressed in a declarative way in order to be easily processed by execution engines and planners (Antonelli et al., 2018). A conceptual model consists of two parts: a domain model and a reasoning model (Trichet and Tchounikine, 1999; Schreiber et al., 1999). The objects of the world (or more exactly of the application domain) are represented in the domain model (similar to an application ontology). The reasoning model describes how an action (task) can be performed. All relevant objects and relationships in the world handled by the reasoning model must therefore be described in the domain model. Domain model are usually described in UML language and implemented with object-oriented languages. The paradigm generally used to represent conceptual models (coming mainly from the artificial intelligence field) is the Task/Method paradigm. This paradigm uses the definitions below.

Table 3
Test cases for Scenario Detect stress

The farmer fails to collect temperature reading, because of failure of the sensors.
The farmer fails to collect humidity reading.
The farmer fails to collect sun radiation reading.
The farmer fails to determine the probability of stressful condition, because of lack to historical data.

Table 4
Scenario Collect temperature

Title: Collect temperature reading from thermal sensors.
Goal: Obtain a unique measure from several sensors.
Context: Greenhouse.
Actors (and agents): Farmer.
Resources: Thermal sensors.
Episodes:
The thermal sensors collect measures about the temperature.
The farmer records all the measures.
The farmer calculates the average of the measures.

Table 5
Test cases for Scenario Collect temperature

The sensors fails to collect temperature reading, because of failure of the sensors.
The farmer fails to record the measures, because of a mistake.
The farmer fails to calculate the average, because of a mistake.

Definition 1. A task is a transition between two families of world states (an action) and is defined by the following fields:

Name the name of the task,
Par list of parameters handled by the task,
Objective task goal,
Methods list of methods performing the task.

Definition 2. A method describes a way (at a single level of abstraction) to perform a task. A method is defined by the following fields:

Header task carried out by the method,
Prec conditions that must be met in order to apply the method.
Effects effects caused by a successful application of the method,
Control description of the order in which subtasks are performed,
subtasks set of subtasks.

Definition 3. A terminal task is a directly executable task. Its execution does not require any decomposition (methods).

The way how the reasoning is done is modelled as a decomposition of tasks and sub-tasks. The reasoning model describes several decompositions of task performance using methods. In this sense, the reasoning model is hierarchical. The decompositions stop to the terminal tasks for which the execution is not described. We have only presented here the fields used in this work. Camilleri et al. (Camilleri et al., 2003) provide a more complete presentation.

3.3. Test Case description using Task/Method model

Scenarios can be represented in the form of a Task/Method model in order to generate of the test cases of scenarios. For example, the Task/Method model for the previous scenarios "Detect stress in crops of tomatoes and peppers" are "Collects temperature reading from" is described in the tables 6 and 7.

Table 6
list of Tasks of "Detect stress in crops of tomatoes and peppers" and "Collects temperature reading from" scenarios

Task: Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) **Methods:** *{*M_1*}*
Task: Collects temperature reading from(Farmer, Thermal sensors) **Methods:** *{*M_2*}*
Task: Collects humidity reading from(Farmer, Humidity sensors) **Methods:** *{}* // terminal task
Task: Collects sun radiation reading from(Farmer, Luminosity sensors) **Methods:** *{}* // terminal task
Task: Determines the probability of(Farmer, A stressful condition) **Methods:** *{}* // terminal task
Task: Collect measures about(Thermal sensors, The temperature) **Methods:** *{}* // terminal task
Task: Records(Farmer, All the measures) **Methods:** *{}* // terminal task
Task: Calculates the average of(Farmer, The measures) **Methods:** *{}* // terminal task

The proposed approach obtains all the combinations of situations described in episodes of the Scenarios. Thus, every task can be considered as a success or failure task. Failure task are related with errors in performing the task (manually) or in the module that implement it. Every episode is represented by a task or a subtask, and the outcome of the episode should be analyzed. If the outcome is successful the execution of the Scenario continues, but in the other case, if the outcome fails, the Scenario ends. In scenarios, the failure cases of scenario achievement are described in the test cases and in task/method model they correspond to a failure execution of subtasks. In the "Detect stress in crops of tomatoes and peppers" and "Collects temperature reading from" scenarios, the correspondence between test cases and subtask execution is as following:

Table 7
list of methods of "Detect stress in crops of tomatoes and peppers" and "Collects temperature reading from" scenarios

Method: M_1 **header:** Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) **Control:** Collects temperature reading from(Farmer, Thermal sensors); Collects humidity reading from(Farmer, Humidity sensors); Collects sun radiation reading from(Farmer, Luminosity sensors); Determines the probability of(Farmer, A stressful condition);

subtasks: *{*Collects temperature reading from, Collects humidity reading from, Collects sun radiation reading from, Determines the probability of*}*

Method: M_2
header: Collects temperature reading from(Farmer, Thermal sensors)
Control:
Collect measures about(Thermal sensors, The temperature);
Records(Farmer, All the measures);
Calculates the average of(Farmer, The measures);
subtasks: *{*Collect measures about, Records, Calculates the average of*}*

For the scenario "Detect stress in crops of tomatoes and peppers":

- **Test case:** The farmer fails to collect temperature reading, because of failure of the sensors:
  - **failure of subtasks:** Collects temperature reading from(Farmer, Thermal sensors)
- **Test case:** The farmer fails to collect humidity reading:
  - **failure of subtasks:** Collects humidity reading from(Farmer, Humidity sensors)
- **Test case:** The farmer fails to collect sun radiation reading:
  - **failure of subtasks:** Collects sun radiation reading from(Farmer, Luminosity sensors)
- **Test case:** The farmer fails to determine the probability of stressful condition, because of lack to historical data:
  - **failure of subtask:** Determines the probability of(Farmer, A stressful condition).

For the scenario "Collects temperature reading from":

- **Test case:** The sensors fails to collect temperature reading, because of failure of the sensors:
  - **failure of subtasks:** Collect measures about(Thermal sensors, The temperature).
- **Test case:** The farmer fails to record the measures, because of a mistake:
  - **failure of subtasks:** Records(Farmer, All the measures).
- **Test case:** The farmer fails to calculate the average, because of a mistake:
  - **failure of subtask:** Calculates the average of(Farmer, The measures).

4. The proposed approach

The proposed approach is a pipeline process of three activities. The first activity is Scenarios' description. In this activities, Stakeholders describes the Scenarios using an application based on a semantic wiki platform. Thus, the stakeholders write narrative descriptions of the Scenarios, and the application provides a semantic representation to enrich the narrative descriptions. The second activity consists in translating the Scenarios described trough narrative text and a semantic formalization into an specification based on Task / Method model paradigm. The translation is performed automatically by the application based on the semantic wiki platform. The application implements some rules that make use of semantic queries in order to obtain the specification based on Task / Method model. This activity is performed automatically by the tool. Then result of the translation, that is, the specification in Task / Method model is described in a standard for sharing ontologies on the web: owl semantic formalization. This formalization describes the Scenarios in a way that they can be computed in order to obtain all the combination in the flow of the execution of the Scenarios. This is the third task of the proposed approach, the identification of all the combination through the execution engine. This is also an automatic activity performed by a different application from the first one. Two different application are combined in this proposal. The first one is an application based on a semantic wiki, that capture the description of the Scenarios in a narrative and semantic way, that also implements the translation to the task / method model specification. Then, the second application uses the specification in task / method model specification to analyse the flow of actions in the scenarios in order to provide all the possible combinations. This combinations is described through an execution tree. If we consider only one Scenario, this execution tree is similar to a decision table regular used in black box testing. Nevertheless, our proposed approach considers several scenarios

which can also be nested, that is, one scenario depends of the result of another scenario. This situation is hard to solve with decision tables. Our approach organizes all the combination in a tree, and a test engineering can traverse the tree to identify the situation that can be interesting to test or can discard the situation that are rare to happen and the effort of testing is not valuable. Figure 1 summarize the approach.

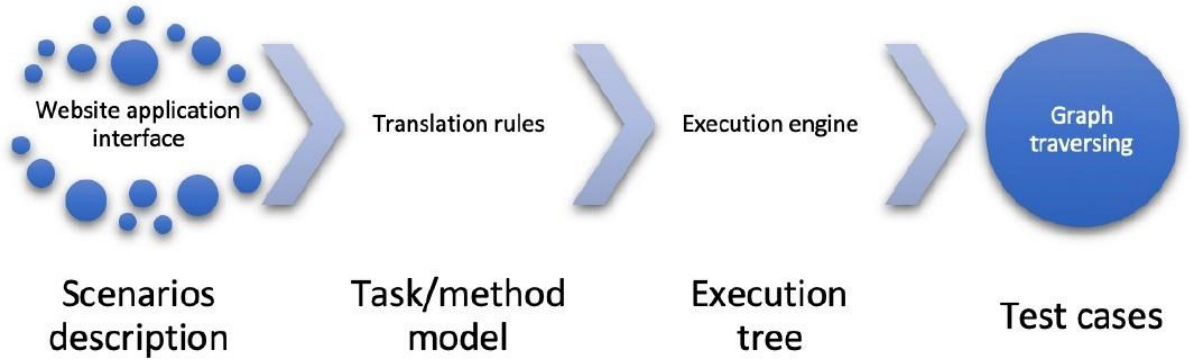

Fig. 1. Test cases generation process

4.1. Scenarios description

This is the first activity where a group of stakeholder collaboratively describes Scenarios in a tool bases on a semantic media wiki. It is important the collaborative work because the knowledge is generally spread in many stakeholder. Moreover, the description of the scenarios is easy because they should be done in a narrative way, and the tool obtain a semantic representation from the narrative description.

The visualization and navigation form of a Scenario is shown in Figure 2. The image describes the Scenario previously detailed in Table 2: It includes the Scenario title as the page title. From the top to the bottom of the page appears: a content summary of the page, and then project, context, and objectives, which are textual information. The actors and resources are symbols represented as concepts in the semantic wiki. Both of them could be selected from a list of previously defined symbols or typed by the user.

Additionally, the user can navigate to any of these symbols by clicking on any of these words. More importantly, either Farmer, Product, Quality levels, and Orders are recognized by the tool as symbols to be used in the derivation tests. A triplet of symbols is the representation of episodes following the LEL definitions (Antonelli et al. (2018)). The episodes can be navigated, and the tool combines them with derivation rules. As the tools are based on Semantic Mediawiki, it includes the functionality to be collaborative edited and to check the editions' history.

The tool, AGUTER, includes a visualization form with a well-organized information description and navigational capabilities. On the other hand, the tool provides an edition form that allows input information with vocabulary suggestions. Indeed, after the information is written in the edition form, and the user saves the changes, it translates the text to semantic information, for example, the application generates typed links among Scenarios and symbols. Some attributes are described with plain texts as context and objective. Nevertheless, some other attributes as actors and Resources are completed with controlled vocabulary tokens because they are LEL elements. The episodes are specific attributes since they are structured in triplets (subject, predicate, object), the form is designed with three fields: to an actor (subject), a verb (predicate), and a resource (object). Finally, several episodes may be part of the same Scenario.

Every element is categorized by the tool either as a Scenario or subject, verb, or object. The category structures the descriptions to derive tests. Additionally, the category is also used to gather the information and present it to the users. They could control and enhance that information. Thus, the tool provides a

page with all the LEL symbols outlined in the tool. This page is dynamic and automatically generated. It includes a query similar that those needed to derive the tests.

4.2. Task Method model / derivation rules

This section describes the proposed rules for translating Scenarios into a conceptual model (tasks and methods). These rules have already been presented in previous works (Antonelli et al., 2018, 2020), so only a summarize is described here.

Rule 1 (Tasks Identification): *In the scenario model, each verb in the Scenario's episodes is translated into a task in Task/Method model. Each Scenario title is also a task in Task/Method model.*
For example, the application of the rule 1 on the scenario "detect stress in crops of tomatoes and peppers" gives :

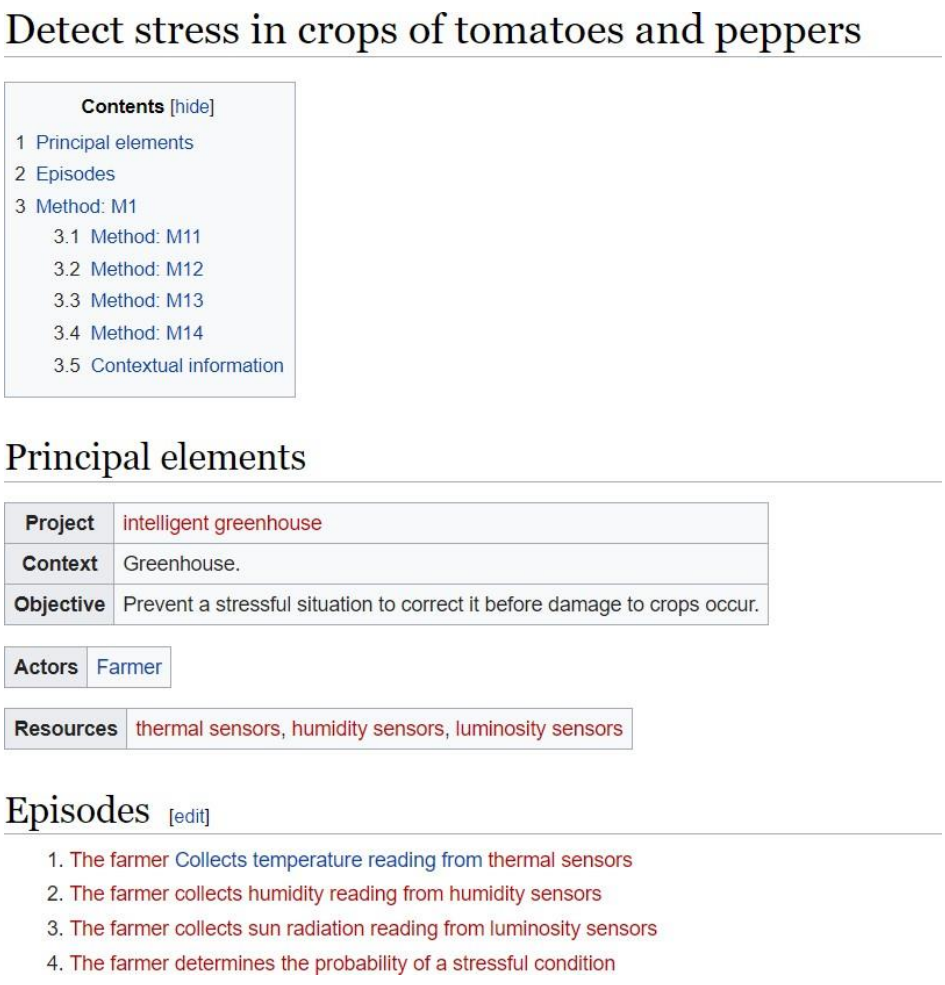

Fig. 2. Scenario navigation form

- Scenario: detect stress in crops of tomatoes and peppers → Task: detect stress
- Episode: The farmer collects temperature reading from thermal sensors → Task: collects temperature reading from

Rule 2 (Task's Parameters Identification): *Each resource of the scenario model linked to an action is translated into a parameter of the task representing the linked action of the Task / Method model.*

For the scenario "detect stress in crops of tomatoes and peppers", the rule 2 generates :

- Episode: The farmer collects temperature reading from thermal sensors → Task: Collects temperature reading from(Farmer, Thermal sensors)

Rule 3 (Scenario's and Episode's method): *The achievements of Scenarios and their episodes are associated to methods in Task / Method model.*

For example:

- Achievement of detect stress in crops of tomatoes and peppers → Method: M1
- Achievement of episode: Collects temperature reading from → Method: M2

Rule 4 (Sequence of tasks): *Different lines in the episodes part of a Scenario is translated by a sequence of tasks in the control part of a method in the Task/Method model.*

For example:

Episodes for the scenario detect stress in crops of tomatoes and peppers:
The farmer collects temperature reading from thermal sensors
The farmer collects humidity reading from humidity sensors
↓
Method: M1
Control
{
Collects temperature reading from(Farmer, Thermal sensors);
Collects humidity reading from(Farmer, Humidity sensors);
}

4.3. Execution Engine

In this work, we will assume that: *the execution of a task can only succeed or fail*. The purpose of the execution engine is to execute the task/method model resulting from the application of the translation rules. From this execution, we want to be able to generate all test cases of scenarios.

The proposed approach is to store all possible executions of a task in a data structure called Execution Tree (ET). Due to the hierarchical nature of task/method models, the execution paths are saved in a tree data structure.

An ET is composed of two types of nodes:

Definition 4. An *etask* node represents an task that have been executed. It has access to the description of the executed task (from the task/method model). Moreover, an *etask* has an execution status which is success or failure (following the previous assumption).

Definition 5. An *emethod* node corresponds to an executed method. It has a link to the executed method of the task/method model) and an execution status: success or failure.

For example, in Figure 3, an ET for the task "Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors,Humidity sensors, Luminosity sensors)" is presented. It contains all possible executions of this task. An ET is a tree which alternates *etask* nodes and *emethod* nodes. In example of Figure 3, there is an alternation between *etasks* represented by a box and *emethods* delimited by an oval. In this Figure, *etasks* and *emethods* with the failure status are shown on a grey background and *etasks* and *emethods* with the success status on a white background. From this tree, there is four possible executions of the task "Collects temperature reading from(Farmer, Thermal sensors)", that are "M2", "M3", "M4" and "M5". "M3" succeeded and the others failed. The *emethod* "M4" failed because "Collect measures

about(Thermal sensors, The temperature)" succeeded but "Records(Farmer, All the measures)" failed. The goal of the execution engine is to generate an ET for a given task.

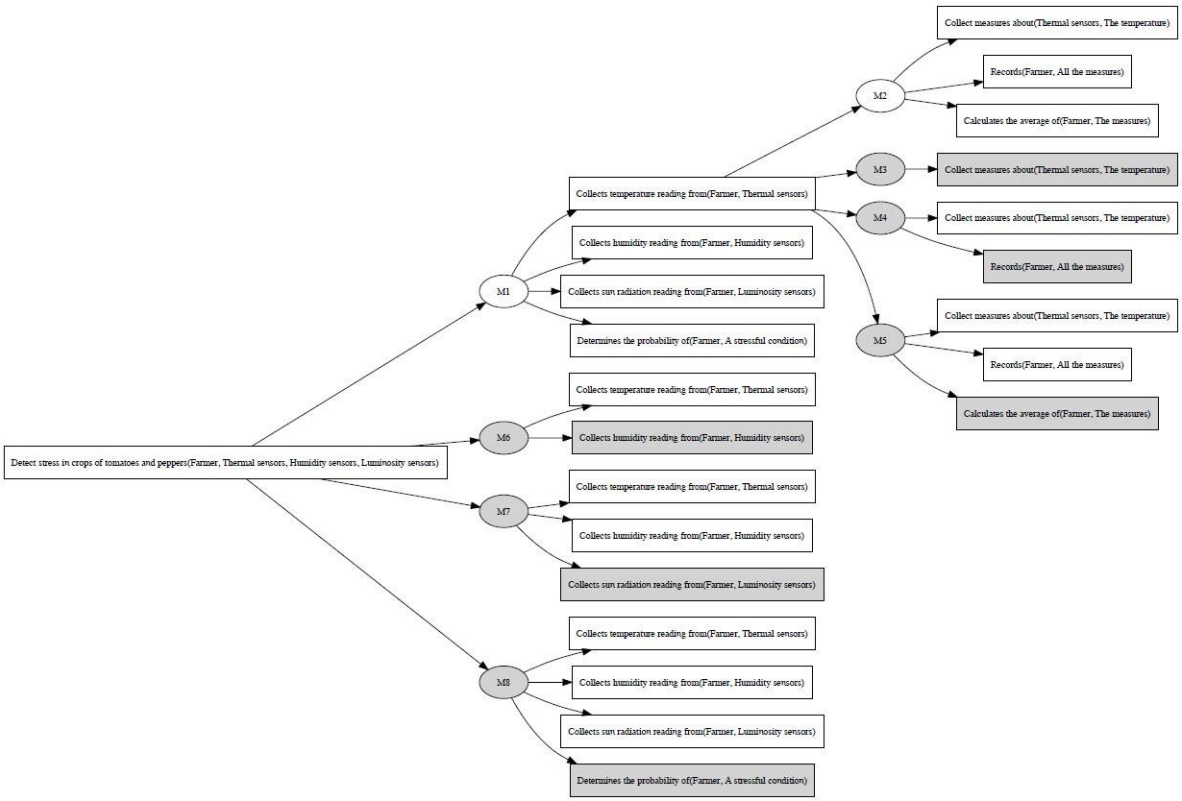

Fig. 3. Execution tree for Detect stress in crops of tomatoes and pepper task

The logic of the execution status assignment is the following : each task having methods owns in its control field the execution order of sub-tasks which achieve this task. The execution status (success or failure) of this task depends thus on the execution status of its sub-tasks. As all task decompositions (through their methods) end on terminal tasks (tasks without methods), therefore, the execution status of all none-terminal tasks is determined by the execution status of terminal tasks. In other words, only terminal tasks directly succeed or fail and this status is propagated to other tasks through their methods. The algorithm of the execution engine is described in Algorithm 1. It generates an ET for a task t with an *emethod* em (at the first call the value of parameter em is initialized to *null*). It returns an *etask* et representing the executions of the task t. The returned *etask* et is thus the root of the ET produced. This algorithm generates first an *etask* et for the task t with a success status, and adds it to the *emethod* em given in parameter. if the task t is terminal then em is duplicated without et (em_1) and et is cloned ($et1$) and added to $em1$. The both em_1 and et_1 have a status to failure. In this way, for terminal tasks, two paths are generated, one with a parent *emethod* (em) and a terminal *etask* (et) with a success status and the other with a parent *emethod* (em_1) and a terminal *etask* (et_1) with a failure status. In the case

where the task t is not terminal then for all methods of the task t which are applicable (having their preconditions satisfied) a new *emethod* (em_2) is generated with an execution status to success, and then the control part of em_2 is launched. It is important to note that the launching of the control part of em_2 will recall the execution engine for each subtask with the *emethod* em_2 as parameter.

Algorithm 1 Execution engine to achieve the task t thanks to an *emethod* em initially set to *null*

```
1:  generate an etask et from the task t with em as parent and the status success
2:  if t is a terminal task then
3:     generate another emethod em1 from em with the status failure
4:     generate an etask et1 from the task t with em1 as parent and the status failure
5:  else
6:     set methods = all methods of t;
7:     for all m in methods do
8:        if preconditions of m are satisfied then
9:           generate emethod em2 from et with the status success
10:          launch the control field of em2
11:       end if
12:    end for
13: end if
14: return et
```

This algorithm was applied to the task "Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors,Humidity sensors, Luminosity sensors)" and produced the ET presented in Figure 3.

4.4. Test cases generator

In this section the test cases generator is presented. The objective of the test cases generator module is to produce all test cases from an ET. Let us consider the ET of the previous example, the *etask* for "Collects temperature reading from(Farmer, Thermal sensors)" has four *emethods* "M2", "M3", "M4" and "M5". The *emethods* "M3", "M4" and "M5" own an *etask* with the failure status which entails the failure status of theses *emethods*. Thus, it exists three failure cases of the *etask* "Collects temperature reading from(Farmer, Thermal sensors)". Theses possible the failure cases of the *etask* "Collects temperature reading from(Farmer, Thermal sensors)" will also cause failure cases for the *etask* "Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)" through the *emethod* "M1". Therefore, failure status is propagated from *etasks* with a failure status to the root of the ET. A test case corresponds thus to an execution path ending by a terminal *etask* with the failure status.

The general algorithm of the test cases generator is described in Algorithm 2. It computes the set of test cases (T Cases) containing all possible test cases for an ET. This algorithm starts by extracting all "terminal" *etasks* of ET with a failure status (F_ETasks). Then, for each failure terminal *etask* et, it generates a path p by traversing the ET from the root to the terminal *etask* et ; and saves this path in the set T_Cases. Let us note that one test case is an alternated sequence of *etasks* and *emethods*.

The Algorithm 2 was applied to an ET for the task "Detect stress in crops of tomatoes and pep-

Algorithm 2 Test Cases generator general algorithm for an Execution Tree ET

1: **set** F_ETasks=*{et* in ET such as *et* is an *etask* for a terminal task *t* with a failure status*}*
2: **set** T_Cases=*{}*
3: **for all** *et* in F_ETask **do**
4: **generate** the path *p* from the root of *ET* to *et*
5: T_Cases=T_Cases *{p}*
6: **end for**
7: **return** T_Cases

pers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)" which has been produced by the execution engine previously presented. The test cases generated by this algorithm are :

- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M1', 'Collects temperature reading from(Farmer, Thermal sensors)', 'M3', 'Collect measures about(Thermal sensors, The temperature)']
- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M1', 'Collects temperature reading from(Farmer, Thermal sensors)', 'M4', 'Records(Farmer, All the measures)']
- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M1', 'Collects temperature reading from(Farmer, Thermal sensors)', 'M5', 'Calculates the average of(Farmer, The measures)']
- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M6', 'Collects humidity reading from(Farmer, Humidity sensors)']
- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M7', 'Collects sun radiation reading from(Farmer, Luminosity sensors)']
- ['Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors)', 'M8', 'Determines the probability of(Farmer, A stressful condition)']

Then the test cases returned by the test cases generator are translated into natural language to be presented to end users. The translation of the test cases provided above are the following:

- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) fails because Collect measures about(Thermal sensors, The temperature) fails.
- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) fails because Collect measures about(Thermal sensors, The temperature) succeeds, but Records(Farmer, All the measures) fails.
- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) fails because Collect measures about(Thermal sensors, The temperature) succeeds, Records(Farmer, All the measures) succeeds, but Calculates the average of(Farmer, The measures) fails.
- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) succeeds,

but Collects humidity reading from(Farmer, Humidity sensors) fails.

- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) succeeds, Collects humidity reading from(Farmer, Humidity sensors) succeeds, but Collects sun radiation reading from(Farmer, Luminosity sensors) fails.
- Detect stress in crops of tomatoes and peppers(Farmer, Thermal sensors, Humidity sensors, Luminosity sensors) fails because Collects temperature reading from(Farmer, Thermal sensors) succeeds, Collects humidity reading from(Farmer, Humidity sensors) succeeds, Collects sun radiation reading from(Farmer, Luminosity sensors) succeeds, but Determines the probability of(Farmer, A stressful condition) fails.

4.5. Approach Implementation

In this section, an implementation of the presented approach called AGUTER is briefly described. Overall, end users enter description of scenarios in the system (web application), moreover, they can edit scenarios (remove, modify, add, list, . . .) and consult the execution tree as well as a narrative expression in natural language of test cases for a given scenario (task). In Figure 4, an overview of the AGUTER architecture is shown. The pipeline process proposed in section 4 corresponds to the following parts of AGUTER (Figure 4):

1. *Scenarios description* (website application interface): modules "Semantic Converter" and "semantic Query",
2. *Task/method model* (translation rules): module "XML Converter",
3. *Execution tree* (execution engine): module "Execution Engine",
4. *Test cases* (Graph traversing): modules "Test case generator" and "Narrative Converter".

First, end users edit Scenarios thanks to a web application which generates a narrative description of them, then the module "Semantic converter" transform the narrative to a semantic model which make it possible to obtain an executable model specified in task/method. This task/method model is transmitted by the "XML Converter" to the "execution engine". The "execution engine" parses the xml files and applies the Algorithm 1 to produce an Execution Tree (ET) for a given tasks provided by users. It is interesting to display for a given task its ET, which gives a complementary view of scenarios (all possible executions). We used the graph visualization software Graphviz (Ellson, 2004) (module "Graphviz renderer") for generating graph visualisation as in Figure 3. The ET is also sent to the module "Test Case Generator" which computes the set of all test cases (see Algorithm 2). The set of test cases is transmitted to the module "Narrative Converter" which gives to users all test cases in natural language.

5. Conclusions and perspectives

In this paper we presented an approach to design User Acceptance Tests from Scenarios using the Task/Method model. This work is a continuation of a previous one, where users describe scenarios through a web application that implemented translation rules to generate the corresponding Task/Method

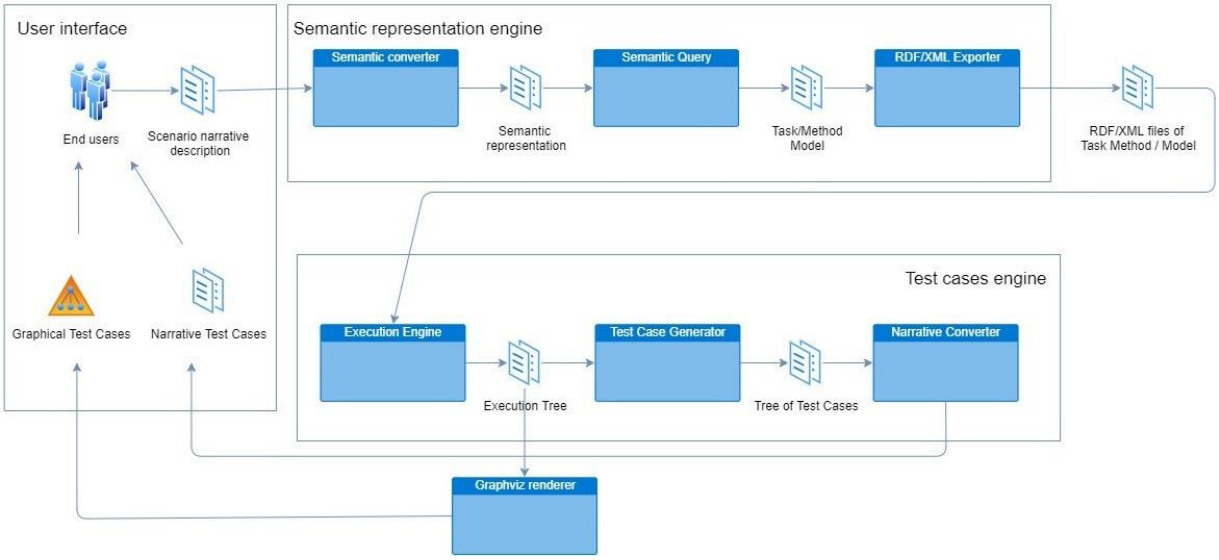

Fig. 4. AGUTER system

model. This paper presented an engine that generates an execution tree of all the traces concerning the possible execution flows in the Scenarios. The application proposed can display the traces in two ways: graphically and textually. From this execution tree, User Acceptance Tests can be extracted. Common approaches to design tests deal with few variables or conditions, while our approach can deal with many. Since the amount can be really big, our tool provides two ways of displaying the tests (concerning the combinations of all the variables). When the number is really big, the graphical way makes it possible to overview all the situations and prune the tree (removing scenarios) if necessary in order to obtain all the tests in a textual way.

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 User Acceptance Tests related to the domain model in the execution engine. Moreover, in this work, we consider only high level of requirements and business processes with many parts expressed in natural language in text form. However, it could be interesting to enrich the description of the scenarios by adding more information about domain objects. For example, in scenarios, we can specify the type and the range of considered values of temperature. In this way, it becomes possible with the same strategy to generate user acceptance tests with a finer level of granularity and extend the applicability of the proposed approach to other types of tests (for example, unit tests or integration tests).

Another important aspect is the practical usability of the proposed approach and its tools. To be useful, software engineers need to be able to use the tools with the least amount of effort. In addition, it could be interesting to improve the visualisation of test cases in order to assist the specification of the future system (with for instance an interactive execution tree, etc.). It would be interesting to visualise the current step of the whole process. It could support developers in their task of testing the software.

Acknowledgments

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

Alexander, I., Maiden, N., 2004. Scenarios, stories, and use cases: the modern basis for system development. *Computing Control Engineering Journal* 15, 5, 24–29.

Ammann, P., Offutt, J., 2016. *Introduction to Software Testing*. (2 edn.). Cambridge University Press.

Andrade, R., Birgin, E.G., Morabito, R., 2013. Two-stage two-dimensional guillotine cutting problems with usable leftovers. Technical Report MCDO15113, Department of Computer Science, Institute of Mathematics and Statistics, University of Sa˜o Paulo, Brazil.

Antonelli, L., Camilleri, G., Grigera, J., Hozikian, M., Sauvage, C., ZARATE´, P., 2018. A Modelling Approach to Generating User Acceptance Tests. In Dargam, F., Delias, P., Linden, I. and Mareschal, B. (eds), *4th International Conference on Decision Support Systems Technologies (ICDSST 2018)*, Springer, Heraklion, Greece.

Antonelli, L., Hozikian, M., Camilleri, G., Fernandez, A., Grigera, J., Torres, D., Zarate´, P., 2020. Wiki support for automated definition of software test cases. *Kybernetes* 49, 4, 1305–1324.

Antonelli, L., Torres, D., Hozikian, M., Hernandez, J.E., 2019. Semantic support for scenarios to improve communication in agribusiness. In *Working Conference on Virtual Enterprises*, Springer, pp. 447–456.

Bjarnason, E., Borg, M., 2017. Aligning requirements and testing: Working together toward the same goal. *IEEE Software* 34, 01, 20–23.

Boucher, M., Mussbacher, G., 2017. Transforming workflow models into automated end-to-end acceptance test cases. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, pp. 68–74.

Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., 2008. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, Association for Computing Machinery, New York, NY, USA, p. 45–48.

Budha, G., Panda, N., Acharya, A.A., 2011. Test case generation for use case dependency fault detection. In *2011 3rd International Conference on Electronics Computer Technology*, Vol. 1, pp. 178–182.

Bystricky´, M., Vranic´, V., 2017. Use case driven modularization as a basis for test driven modularization. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pp. 693–696.

Camilleri, G., Soubie, J.L., Zalaket, J., 2003. TMMT: Tool Supporting Knowledge Modelling. In *7th International Conference on Knowledge-Based Intelligent Information and Engineering Systems, KES 2003, Oxford UK, 03/09/03-05/09/03*, Springer. Pages de la publication : 45-52,partI.

Capretz, L.F., Waychal, P., Jia, J., Varona, D., Lizama, Y., 2019. Studies on the software testing profession. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, IEEE Press, p. 262–263.

Chapra, A.B., Ati, C.D., 2004. Business ethics in islamic context: perspective of a muslim business leader. *Business Ethics Quarterly* 3, 4, 211–221.

Chatterjee, R., Johari, K., 2010. A prolific approach for automated generation of test cases from informal requirements. *SIGSOFT Softw. Eng. Notes* 35, 5, 1–11.

Congram, R.K., 2000. Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimisation. Ph.D. thesis, University of Southampton.

Dos Santos, E.C., Vilain, P., Hiura Longo, D., 2018. Poster: A systematic literature review to support the selection of user acceptance testing techniques. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 418–419.

Ellson, J., 2004. Graph visualization software.

Enoiu, E., Tukseferi, G., Feldt, R., 2020. Towards a model of testers' cognitive processes: Software testing as a problem solving approach. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 272–279.

Entin, V., Siegl, S., Kern, A., Reichel, M., Meyer-Wegener, K., 2009. A scenario-centric approach for the definition of the formal test specifications of reactive systems. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, pp. 179–183.

Garousi, V., Elberzhager, F., 2017. Test automation: Not just for test execution. *IEEE Software* 34, 2, 90–96.

Hsieh, C., Tsai, C., Cheng, Y.C., 2013. Test-duo: A framework for generating and executing automated acceptance tests from use cases. In *2013 8th International Workshop on Automation of Software Test (AST)*, pp. 89–92.

Hsieh, C.Y., Tsai, C.H., Cheng, Y.C., 2013. Test-duo: A framework for generating and executing automated acceptance tests from use cases. In *2013 8th International Workshop on Automation of Software Test (AST)*, pp. 89–92.

Hussain, A., Nadeem, A., Ikram, M.T., 2015. Review on formalizing use cases and scenarios: Scenario based testing. In *2015 International Conference on Emerging Technologies (ICET)*, pp. 1–6.

IAGCargo, 2013. https://www.iagcargo.com/iagcargo/portlet/en/html/fleetuld.

ISO/IEC/IEEE, 2014. Ieee draft international standard for software and systems engineering–software testing–part 4: Test techniques. *ISO/IEC/IEEE P29119-4-DISMay2013* pp. 1–132.

Jackson, M., 1995. The world and the machine. In *1995 17th International Conference on Software Engineering*, pp. 283–283.

Jeeva Padmini, K., Perera, I., Dilum Bandara, H.M.N., 2016. Applying agile practices to avoid chaos in user acceptance testing: A case study. In *2016 Moratuwa Engineering Research Conference (MERCon)*, pp. 96–101.

Khamaiseh, S., Xu, D., 2017. Software security testing via misuse case modeling. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pp. 534–541.

Lei, H., Wang, Y., 2016. A model-driven testing framework based on requirement for embedded software. In *2016 11th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pp. 1–6.

Lenstra, J., Rinnooy Kan, A., 1979. Complexity of packing, covering, and partitioning problems. In Schrijver, A. (ed.), *Packing and Covering in Combinatorics*. Mathematisch Centrum, Amsterdam, pp. 275–291.

Lipka, R., Potua´k, T., Brada, P., Hnetynka, P., Vina´rek, J., 2015. A method for semi-automated generation of test scenarios based on use cases. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 241–244.

Lizhe Chen, Qiang Li, 2010. Automated test case generation from use case: A model based approach. In *2010 3rd International Conference on Computer Science and Information Technology*, Vol. 1, pp. 372–377.

Mansour, N., Sleiman Haidar, G., 2010. Parallel Metaheuristic Algorithm for Exam Timetabling. In *6th International Conference on Natural Computation*, IEEE, pp. 471–475.

Masuda, S., 2017. Software testing design techniques used in automated vehicle simulations. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 300–303.

Miao Huaikou, Liu Ling, 2000. A test class framework for generating test cases from z specifications. In *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000*, pp. 164–171.

Monpratarnchai, S., Fujiwara, S., Katayama, A., Uehara, T., 2013. An automated testing tool for java application using symbolic execution based test case generation. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, Vol. 2, pp. 93–98.

Nogueira, S., Sampaio, A., Mota, A., 2014. Test generation from state based use case models. *Formal Aspects of Computing* 26, 3, 441–490.

Nomura, N., Kikushima, Y., Aoyama, M., 2014. A test scenario design methodology based on business context modeling and its evaluation. In *2014 21st Asia-Pacific Software Engineering Conference*, Vol. 1, pp. 3–10.

d. P. Leite, J.C.S., Franco, A.P.M., 1993. A strategy for conceptual model acquisition. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 243–246.

Pandit, P., Tahiliani, S., Sharma, M., 2016. Distributed agile: Component-based user acceptance testing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–9.

Pandit, P., Tahiliani, S., Sharma, M., 2016. Distributed agile: Component-based user acceptance testing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–9.

Philip, G., Dsouza, M., Abidha, V.P., 2017. Model based safety analysis: Automatic generation of safety validation test cases.

In *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, pp. 1–10.

Ramler, R., Klammer, C., 2019. Enhancing acceptance test-driven development with model-based test generation. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 503–504.

Ryan, D., 1995. Solving truck problems, Proceedings of the 31st Annual Conference of the ORS of New Zealand, August 31 September 1, Willington, New Zealand, pp. 165–172.

Sarmiento, E., Leite, J.C., Almentero, E., Sotomayor Alzamora, G., 2016. Test scenario generation from natural language requirements descriptions based on petri-nets. *Electronic Notes in Theoretical Computer Science* 329, 123 – 148. CLEI 2016 - The Latin American Computing Conference.

Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N.R., Van de Velde, W., Wielinga, B.J., 1999. *Knowledge Engineering and Management: The CommonKADS Methodology*. The MIT Press.

Stoyanova, V., Petrova-Antonova, D., Ilieva, S., 2013. Automation of test case generation and execution for testing web service orchestrations. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pp. 274–279.

Svensson, R.B., Regnell, B., 2015. Aligning quality requirements and test results with quper's roadmap view for improved high-level decision-making. In *2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing*, pp. 1–4.

Takagi, T., Noda, K., 2016. Partially developed coverability graphs for modeling test case execution histories. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pp. 1–2.

Trichet, F., Tchounikine, P., 1999. Dstm: a framework to operationalise and refine a problem solving method modeled in terms of tasks and methods. *Expert Systems with Applications* 16, 2, 105 – 120.

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J., 2006. Automation of gui testing using a model-driven approach. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, Association for Computing Machinery, New York, NY, USA, p. 9–14.

Williams, K. Jr., 1981. *Behavioural Aspects of Marketing* (2nd edn.). Butterworths-Heinemann, London.

Young, R., 2004. *The requirements engineering handbook* (1st edn.). Artech House Publishers.