JOURNAL OF DECISION SYSTEMS
https://doi.org/10.1080/12460125.2023.2229579

Taylor & Francis
Taylor & Francis Group

User acceptance test for software development in the agricultural domain using natural language processing

Leandro Antonelli[a], Guy Camilleri[b], Diego Torres[a,c] and Pascale Zaraté[b]

[a]LIFIA - CICPBA - Facultad de Informática, Universidad Nacional de La Plata, La Plata, Argentina; [b]IRIT Université de Toulouse, CNRS, Toulouse, France; [c]Depto. CyT, Universidad Nacional de Quilmes, Bernal, Argentina

ABSTRACT
Software test case design is one of the most challenging activities since many actors with different backgrounds must cover most of the user's needs and expectations. In the agricultural domain tasks can be done in very different ways since practices vary worldwide. Thus, in this context, it is very hard to design test cases to validate requested functionality that automatizes some farm tasks. This paper proposes an approach to make the testing step easier, designing User Acceptance Tests (UATs) from requirements captured through scenarios. The scenarios capture the knowledge of different stakeholders (farmers) and using natural language processing tools, the approach proposed to consolidate the set of scenarios in a consistent and coherent base of knowledge organised in a tree, from where the design of test cases is extracted using the Task/Method model, a tool from the Artificial Intelligence.

ARTICLE HISTORY
Received 31 October 2022
Accepted 21 June 2023

KEYWORDS
User Acceptance Tests; Requirements specifications; Scenarios; Task/Method model; Agriculture production systems

1 Introduction

Software development is a big challenge considering the communication gap that arises between the IT team and the clients since these two groups of people speak different languages. The IT team speaks a technical language oriented to the design of the solution that the software system should provide, while the clients speak a technical language oriented to the problem domain. Thus, it is very hard for both parties to communicate with each other. This communication is even more difficult because inside each group there are many people with different backgrounds, knowledge, and language. For example, there are sponsors, users, and domain experts from the client side, while there are commercial officers, managers, analysts, and developers from the IT team. Moreover, the situation is even more complex in the agricultural domain, because agricultural practices are mainly regional since weather, soil conditions, plants, materials, and techniques are specific to a region. For example some farmers produce in an organic way while others produce in a conventional way. That is, while organic farmers are focused on ecological production, conventional farmers generally use modern technologies like chemical crop protection and synthetic fertilisers among other practices (Abeysiriwardana et al., 2022;

CONTACT Leandro Antonelli lanto2004@gmail.com LIFIA - CICPBA - Facultad de Informática, Universidad Nacional de La Plata, 50 y 120, La Plata 1900, Argentina

© 2023 Informa UK Limited, trading as Taylor & Francis Group

Niazi & Ghafoor, 2021; Saah et al., 2022). Then, water resources of the region (because of the weather conditions or because of its proximity to sources of fresh water) can restrict the type of watering practices that can be carried out. For example, if there are plenty of water resources, a sprinkler irritation technique can be carried out (Shruthi et al., 2017). If the resource of water is limited a drip irrigation system is a better option (Yang-Ren & Zhi-Wei, 2016). Finally, hydroponic farming is a technique to use where water must be saved and the quality of the soil is not good (Prince et al., 2022). Although every application domain has its particularities and a proper language, the agriculture field combines different elements: living beings (the plants), different levels of technology (from manual practices to complete automatisation with IoT), different conditions (from economic conditions to natural conditions), and many stakeholders involved (the chain supply in the agriculture include management, production, commercialization, etc.). These elements create a variety in the language that could be similar to the Health application domain, although in the Health field, fewer people than in the agricultural domain participate. Thus, although the proposed approach is domain independent, the agriculture domain includes a variety and complexity in the language to justify the applicability and usability of the proposed approach. The amount and variety of stakeholders are also related with a specific context of development, the market-driven software development. In this type of development, managing requirements is a big issue since it is not easy to elicit and agree on requirements from all the stakeholders (Karlsson et al., 2007). Thus, designing test cases (Budha et al., 2011) is also a great challenge.

Let's consider the following situation that involves a farmer who knows how to grow crops and a business administrator who knows how to deal with economic accounts. The farmer says to the administrator: 'Can you help me to sell more products because I need to increase my benefits?'. And the business administrator answers 'Increasing the selling of goods is not the only way to increase the utilities'. It seems that they are talking about different things, but in fact, they use different words to express the same idea and it could seem confusing for someone not familiar with the terms, because 'goods' are materials that satisfy human wants and provide 'utility'.

Communication of both worlds is extremely important for the project software development's success since the clients state their goals, needs, and wishes that should be translated into requirements. And these requirements (as for example in the software development V model (Forsberg & Mooz, 1991)) are usually the source information to design the test cases that assure the software application satisfies the client's need.

Traditional techniques to design test cases can be grouped into two main categories. White box tests and black box tests. The first one needs a detailed specification of execution workflows of the software application to analyse all possible flows to test all of them. The second type of category needs all the data that a software application needs to provide all the possible combinations concerning the data to perform a decision table that considers every possible combination. Thus, in order to test a software application, generally source code is used as input, although it is beneficial to use an artefact of early stages as requirements so the validation of the software application is tied closely to clients' products. This type of test is called system test and user acceptance test, both types of tests have the same goal, to ensure that the whole system satisfies clients' needs. The difference between both types relies on that the person who performs the tests: user acceptance tests are performed by some

representative of the client, while system tests are performed by some member of the development team (ISO/IEC/IEEE, 2010). Generally, system and user acceptance tests, are more related to white-box tests, where a flow of actions is tested. In this paper, two expressions are used 'User Acceptance Tests' and 'Test cases'. It is important to mention that 'User Acceptance Test' is a specific type of test as for example, others specifics types of tests 'System Usability Tests', and 'Integration Tests'. Meanwhile, 'Test case' is a generic expression that refers to a set of actions or instructions that validates a specific aspect of a product.

Scenarios (Alexander & Maiden, 2004) are a suitable type of artefact to specify knowledge of the domain. That is, they can be used to describe the dynamic of a system as well as the requirements of a software system. Scenarios are described using natural language, without introducing complex formalism, so they are adequate to be produced and consumed by the client. It implies removing the aspects that are not relevant to analyse the test cases. Moreover, a scenario describes a sequence of events, and this is related to the flows of actions that lately the software application will provide. Furthermore, scenarios usually describe situations, materials, and actors that are related to conditions and data that lately will appear in the source code. Thus, a scenario provides in an early stage a good description of a software application that can be used as the input that the test cases design activity needs.

Designing test cases conceptually requires processing the flows of execution or the set of data to obtain a set of combinations of them, to evaluate the software application with the wide range of possible situations to assure that its behaviour is the intended behaviour (that is, it agrees with the requirements). Task /Method model (Trichet & Tchounikine, 1999) is a technique of the Artificial Intelligence that provides the capability to process semi structured descriptions (for example scenarios) to obtain a weaving (combinations). Thus, the Task/Method model can be used to process Scenarios to obtain all the combinations.

The goal of this paper is to provide an approach to relate or compose scenarios and use the Task/Method model to obtain test cases without inconsistencies inserted by vocabulary concerns. Moreover, the approach also considers excluding any irrelevant aspects or cases when selecting test cases for analysis. That is, the approach uses the Scenarios as input, and produces test design as output. The approach does not consider any feedback to correct the Scenarios, it only processes the scenarios (with some manual and some automatic tasks) and produces the test design. These tests can either be used as a framework to specify more detailed user acceptance or system tests.

Nevertheless, this proposal extends the previous publication regarding the analysis of the scenarios. The scenarios are written by different people, that is a farmer can write one scenario while a business administrator writes another one. Although they use different languages, the scenarios need to be related to each other to provide a consistent description of the whole system so that the test cases can cover it. Thus, this new proposal uses natural language processing techniques and semantic support to relate the scenarios, while the previously proposed approach considered that the scenarios were described and consolidated, so they were described using a consistent language. We believe that this contribution (the use of natural language processing and semantic support) to relate scenarios is one of the greatest advantages regarding other proposals. At the moment of writing this article and according to authors knowledge there were no

evidence of other proposals to derive test from scenarios using these techniques to consolidate the scenario.

This approach is intended to be used mainly in an agile software development process where the specification of requirements is not so detailed, thus scenarios can add more information. Moreover, since agile software development is incremental, our proposed approach provides continuously the whole landscape of the test designs sprint after sprint considering the addition of new functionality. Finally, it is important to mention that the proposed approach provides a guideline of the chain of situations that are relevant to test, although it does not provide specific context and detailed results for the tests. That is, our test cases are nearer to user experience than to unit tests. Because of the nature of the agile development, the effort invested in the specification is not so big, thus some vocabulary concerns can arise due to the involvement of different stakeholders. Thus, our proposed approach helps with this issue. Moreover, since User Stories describes requirements vaguely and the scenarios describe the dynamic of the domain (some part automatised by the software application), our proposed approach helps to cope with this issue about the boundaries of the application in the Scenarios. The product owner (or the engineer in charge) is the intended role to use our proposed approach to obtain the test cases to provide to the development team. Our approach uses the Task/Method paradigm in order to plan all possible tests to be done for the developed software (Camilleri et al., 2003).

This paper is organised in the following way. Section 2 discusses some other approaches proposed by other authors for a similar situation. Section 3 describes some concepts needed to understand our proposed approach. Section 4 explains our approach. Finally, Section 5 provides some conclusions.

2 Background

This section describes the two main modelling techniques used in our proposed approach. The first technique is a template for describing the Scenarios: the input of our approach. The second technique is the Task/Method model, a conceptual model that provides the execution capability of the specification of the Scenarios to obtain the execution tree of all the alternative workflows.

2.1 Scenarios

The Scenarios are adequate artefacts to describe the behaviour of an application domain. They simply tell a story, and it is easy for no technical people to do that since everyone knows how to tell a story (for example a joke or an anecdote).

Scenarios can be used in different software development stages, from refining business processes to describing requirements (Alexander & Maiden, 2004). There is a gap between the context domain (the real world where the process is carried out) and the software application (the tool to be used in the real world) (Jackson, 1995). Scenarios can describe both: real-world events like business processes and requirements.

There is a wide range of descriptions of Scenarios from visual artefacts such as storyboards to structured text (Young, 2004). Leite et al. (Do Prado Leite et al., 2000) propose a textual Scenario with a template containing the following attributes: (i) a title that

identifies the scenario, (ii) a goal with the objective that the scenario pursues, (iii) a context that states the starting point of the scenario, (iv) the actors, that is, the subjects that perform actions, (v) the resources, that is, the material and information the actors need, and (vi) the episodes, the sequence of actions to be carried out within the scenario. Table 1 summarises the structure. It is important to mention that the template proposed by Leite et al. (Do Prado Leite et al., 2000) also includes exceptions and restrictions from which non-functional requirements can be obtained as well as more alternative situations. Moreover, the context attribute can include specific information like geographic location, temporary location, and precondition. Nevertheless, this information is not considered in our proposed approach, we plan to use it in a further work.

Let's consider, for example, the domain of agriculture where two different farmers describe the activity of fertilising. Fertilising is performed usually while watering since the fertilisers are dissolved into water. Thus, big farms have an infrastructure to perform watering with pipes, pumps, and tanks. The fertilisers are poured into the tank, the pump is turned on, and the mixture of water with the fertilisers is spread over the field. This situation is described in Table 2. Nevertheless, fertilisation can be done in another way if the field does have not the described infrastructure or if the field has the infrastructure but the water in the tanks is not enough to activate the pump. In these situations, fertilisation is performed manually using a spraying backpack. This alternative scenario is described in Table 3.

Another important characteristic of the scenarios is that they can be composed, that is, one episode of one scenario can be described as a whole scenario. For example, fertilising is one of the main activities to perform during the cultivation, but there are others: watering, performing cultural labours (removing weeds among the plants), etc. Thus, let's consider a scenario that describes the activities during the vegetative phase of the plant. This scenario includes different steps in their episodes. One of the steps is 'fertilise using the irrigation pipe', Table 4. Thus, the detail necessary to understand this activity must be obtained from the previous scenario Table 2.

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: These are the steps that actors execute to reach the goal.

Table 2. Scenario Fertilize using the irrigation pipe.

Title: Fertilize using the irrigation pipe.
Goal: Add nutrients to the plant.
Context: The cistern has enough water to activate the irrigation pipe.
Actors (and agents): Farmer.
Resources: cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals.
Episodes: The farmer calculates the amount of minerals. The farmer dilutes the minerals in the water. The farmer pours the mixture into the irrigation pipe. The farmer activates the irrigation pipe. The farmer pours fresh water into the irrigation pipe.

Table 3. Scenario Fertilize using the spraying backpack.

Title: Fertilize using the spraying backpack.
Goal: Add nutrients to the plant.
Context: The cistern does not have enough water to activate the irrigation pipe.
Actors (and agents): Farmer.
Resources: water, backpack, minerals, chart to calculate the amount minerals.
Episodes: The farmer calculates the amount of minerals. The farmer dilutes the minerals in the water. The farmer pours the mixture into the backpack. The farmer sprays the liquid into the plant.
The farmer washes the backpack.

Table 4. Scenario Cultivate the plants.

Title: Cultivate the plants.
Goal: Perform the necessary activities to foster the plant to grow up.
Context: The plant has reached its vegetative phase.
Actors (and agents): Farmer.
Resources: water, fertilizer, pruning scissors.
Episodes:
The farmer waters the plants.
The farmer performs cultural labor.
The farmer fertilizes using an irrigation pipe.

It can be noticed, that the first two episodes of the scenario in Table 4 are too general (″The farmer waters the plants″ and 'The farmer performs cultural labours'), while the last episode (″The farmer fertilises using the irrigation pipe″) is very specific. This example was provided to show explicitly the link between the Scenario 'Cultivate the plants' and 'Fertilises using the irrigation pipe'. Nevertheless, the episodes are commonly more general as 'The farmer fertilises the plants', since this is the activity that he performs. Then, the activity can be performed in two ways: 'Fertilise using the irrigation pipe' and 'Fertilise using the spraying backup'.

2.2 Task/Method model

The Task/Method model is a conceptual model where knowledge is described in declarative form. This facilitates their processing by execution engines and planners (Antonelli et al., 2018). A conceptual model is composed of two sub-models: a domain model and a reasoning model (Adla et al., 2007; Schreiber et al., 1999; Trichet & Tchounikine, 1999). The domain model contains the objects of the world (or more precisely the application domain, similar to an application ontology). The achievement descriptions of tasks are described in the reasoning model. Therefore, all relevant objects and relations of the world used in the reasoning model must be represented in the domain model. Generally, the UML modeling language is used to describe domain models. They are often implemented using an object-oriented language. The Task/Method paradigm (coming mainly from the field of artificial intelligence) is generally used to represent models of reasoning. This paradigm is defined below.

Definition 1. *A task (or an action) is a transition between two world states. The following attributes define it:*

Name: *the name of the task,*
Par: *the list of parameters used by the task,*
Objective: *the task goal,*
Methods: *the list of methods that enable the task to be performed.*

Definition 2. *One way (at a single level of abstraction) to perform a task is described in one method. The method is defined by :*

Header: *task achieved by the method,*
Prec: *conditions that must be satisfied in order to apply the method,*
Effects: *effects caused by a successful application of the method,*
Control: *description of the order in which the subtasks should be executed,*
subtasks: *set of subtasks.*

In a reasoning model, the reasoning is modelled by the decomposition of tasks into subtasks. One decomposition is represented using one method. The reasoning model is therefore a set of hierarchical decompositions of tasks. Decompositions end with the terminal tasks which are directly executable (there is no method for them).

This section describes briefly the attributes required to understand the proposed approach. A complete description can be found in Camilleri et al. (Camilleri et al., 2003). In addition, the precondition and effect fields will not be used in this work. For the Task/Method model, we deal only with a high-level description that does not require a domain model.

3 Approach

3.1 Our approach in a nutshell

The proposed approach is a three steps approach that uses Scenarios as input and produces a set of test cases as output. The first step of the approach consists in integrating the scenarios in a tree, where the relationship used to build a tree is 'an episode is described as a scenario'. This step is performed automatically using natural language processing and semantic tools, to discover the relationships between episodes and scenarios. Then, the second step of the approach consists in removing the branches of the tree that are not relevant to analyse to design test cases. This step is performed manually since the engineer in charge of the development of the software application based on the scenarios is the one who knows the scope of the system. Finally, the third step of the approach consists in creating all the combinations of the possible flows of the sequence of actions to design the test cases. This step is performed automatically using the Task/Method model. Figure 1 shows a summary of the approach.

The following subsections describe each step of the proposed approach.

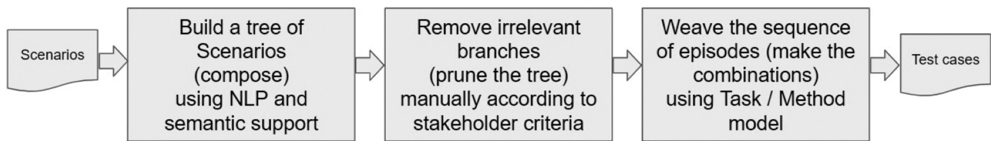

Figure 1. Our approach in a nutshell.

3.2 Step 1, build a tree of scenarios (composition)

The linking between episodes and scenarios can face different situations, from the simplest one, where the same expression is used in the episode and the title of the scenarios, to the most complex one, where the expressions are different and it is necessary to analyse in detail the attributes that constitute both different activities to infer whether they are similar. Thus, this step of building a tree by linking scenarios (in fact, linking episodes to scenarios) can be done using three different techniques: (i) syntactic recognition, (ii) semantic recognition, and (iii) analysis of attributes. The rest of this subsection describes each one of the techniques.

It is very important to mention that the composition is done between one episode and one scenario. That is, the proposed approach does not consider the situation where one episode can be linked to several scenarios at the same time. This situation is considered future work.

The first technique, syntactic recognition is the simplest situation and it can be solved using natural language processing tools. Let's consider the scenario 'Cultivate the plants' (Table 4). This scenario includes the episode 'The farmer fertilises using the irrigation pipe'. This expression is quite similar to the one used in the title of the scenario 'Fertilise using the irrigation pipe'. The last part of the expression is exactly the same as 'using irrigation pipe'. The episode includes the subject 'The farmer', and both expressions contain the verb 'fertilize'. The difference regarding the verbs is that the expression that contains the subject 'The farmer' contains the verb conjugated 'fertilises' for the third person, while in the other case, it is written in its bare infinitive 'fertilise'. This situation could be even more complex since the episode of the scenario 'Cultivate the plants' (Table 4) could only state 'The farmer fertilises the plants'. In this situation, this episode should be linked to both scenarios simultaneously. It should be linked to the scenario 'Fertilise using the irrigation pipe' (Table 2) and the scenario 'Fertilise using the spraying backpack' (Table 3). In this case, the link relies mainly on the verb 'fertilise'. In order to perform this syntactic analysis to link episodes with scenarios, we use different tools provided by natural language processing tools. The first one is the Levenshtein distance for syntactic similarity. This technique measures how similar are both expressions regarding characters in common and difference and their relative positions. Using this technique, the distance between 'The farmer fertilises using the irrigation pipe' and 'Fertilise using the irrigation pipe' would be close. The second tool we use is Part of Speech tagging in combination with lemmatisation. Part of a Speech (POS) tagging is a tool that determines the function of every word in a sentence. Thus 'The farmer fertilises the plant' after a pos tagging will provide that: 'the' is a determinant article, 'farmer' is a noun, 'fertilises' is a verb, etc. Since the most significant word in the episode and in the title of the scenarios are the verbs, because they describe the activity, with POS tagging the word 'fertilises' can be identified

from the episode and the word 'fertilise' can be obtained from the title of the scenario. Then, using lemmatisation verb 'fertilises' is transformed to its bare infinitive 'fertilise'.

The second technique, semantic recognition, applies when the expressions are different but the meaning is the same. This analysis is performed using dictionaries and considering synonyms, as well as hyponyms and hypernyms. For example, 'Tomato' is the usual name that every regular consumer uses when shopping in the grocery store. And 'Solanum Lycopersicum' is the scientific name that engineer use. Both 'Tomato' and 'Solanum Lycopersicum' are synonyms. Then, the 'Tomato' is a 'plant'. Thus, 'plant' is the hypernym, while 'Tomato' is the hyponym. Moreover, 'solanum' is the hypernym, and 'Solanum Lycopersicum' is the hyponym. However, 'solanum' and 'plant' are not synonyms. There are glossaries (dictionaries, vocabularies) that describe concepts and also include relationships between them (for example the three mentioned: synonym, hypernym and hyponym)[5].

Let's consider the episode 'The farmer waters the plants' of the scenario 'Cultivate the plants' (Table 4). And consider another scenario with the title 'Irrigate the tomatoes'. The expressions 'The farmer waters the plants' and 'Irrigate the tomatoes' are related since water and irrigate describe the same activity, and tomatoes are one specific type of plant. Using semantic recognition and some POS tagging tools it is easy to find the link between both expressions. With POS tagging the verb 'water' and the noun 'plant' can be identified from the episode. Then, using the same POS tagging, the verb 'irrigate' and the noun 'tomato' can be obtained from the second expression. Then, using some glossary can be identified that 'water' and 'irrigate' are synonyms, while there is a relationship of hyponym and hypernym between 'tomato' and 'plant'. Thus, we can conclude that both expressions are similar.

The last technique: analysis of attributes, applies when it is necessary to determine whether an episode is related to a scenario by analysing the attributes of the scenario (and not only its title). That is, by analysing the descriptions of the scenarios: goal, context, actors, resources, and episodes. For example, fertilising a plant implies using chemical products to add nutrients to a plant. Thus, the episode of one scenario can state: 'The farmer fertilises the plant to add nutrients'. And there is another scenario with the title 'Adding chemical products' and the goal 'to add nutrients'. In this situation, the link of the episode to the scenario can be done through the goal.

Figure 2 shows an example of two scenarios composed. The first two levels represent the scenario of Table 4. That is, 'Cultivate the plants' is the title of the Scenario, while its children are its episodes. Particularly, 'The farmer fertilises using the irrigation pipe', is a child (an episode) of 'Cultivate the plants', but at the same time is another Scenario (Table 2). Thus, his children are its episodes.

3.3 Step 2, remove irrelevant branches (prune the tree)

This step is an entirely manual step that should be performed by the engineer in charge of the software developer who knows the scope of the system. Thus, [5] http://wordnetweb.princeton.edu/perl/webwn https://agrovoc.fao.org/browse/agrovoc/en/ she/he can identify the boxes of the tree that are not relevant for designing test cases, since those activities are outside of the boundaries of the software application. For example, Figure 3 shows an example where the engineer states that two activities will remain

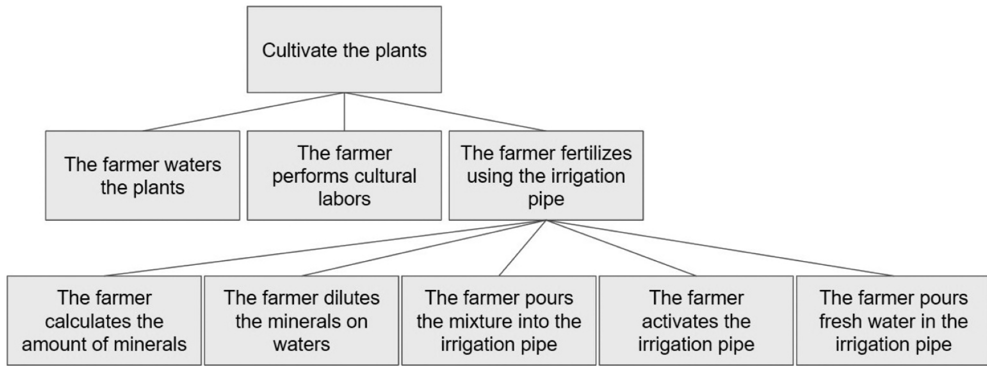

Figure 2. Tree depicting the composition of two scenarios.

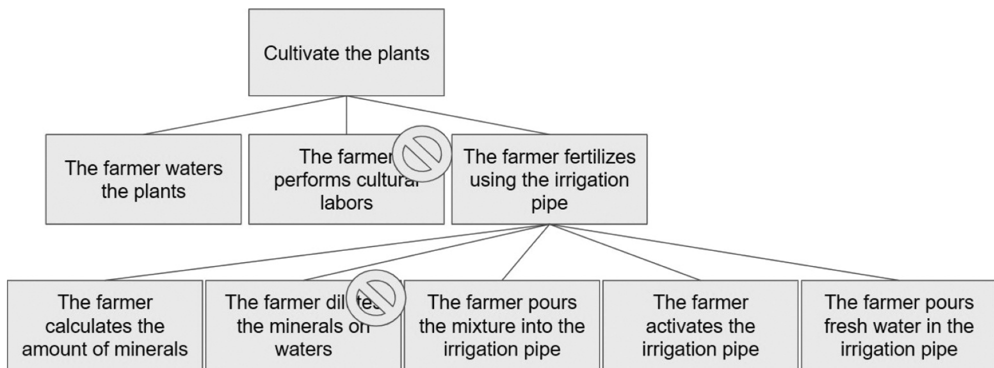

Figure 3. Tree where manual activities are pruned.

outside of the boundaries of the software application. The cultural labour is related to removing the weed and this will keep manually. Nevertheless, the watering will be completely automatised (since sensors will assess the conditions and the software application will turn the machine on when necessary). Then, fertilisation will be almost completely automated. That is, the calculation of minerals will be solved by the software application. Then, the task of pouring the minerals into the tank will remain manually. Finally, the last three tasks: pouring the mixture, activating the pump, and flushing the pipe will be automatised. Thus, it makes sense to consider these steps for designing test cases.

3.4 Step 3, create all the combinations

This step is automatised using task/method model. It consists in converting the tree of scenarios into a task/method model, in order to execute it to provide all the combinations of the relevant test cases. The rest of the subsection is organised in the following way. First, the relation between test cases and the execution of Task/method model is presented. Then we describe how Task/Method model can be built from scenarios. Finally, we explain the generation of test cases.

3.5 The relation between test cases and Task/Method model execution

Let us consider the previous scenarios in the tree of Figure 2 'Cultivate the plants' and 'The farmer fertilises using the irrigation pipe'. A task/method model for these scenarios is shown in Tables 5 and 6.

In our approach, we consider that test cases correspond to situations where an action (task) is performed correctly or not. If an action is executed correctly, we will say that the action was successful, which means more precisely that its execution was successful. Conversely, when an action is not performed correctly, we will consider that it has failed. The execution of each task can therefore be only a success or a failure. Each episode is modelled by a task, and the result of the episode (task) execution must be taken into account. If the result is positive, the execution of the scenario continues, but otherwise, if the result fails, then the scenario ends.

In the scenario paradigm, test cases are related to failure and success cases. In the task/method model, they correspond to failure and success of the execution of subtasks. In the scenarios 'Cultivate the plants' and 'The farmer fertilises using the irrigation pipe', the scenarios test cases and their translation in a task/method model execution are:

For the scenario 'Cultivate the plants':

- **Test case**: The farmer cultivates the plants by watering them and then fertilising them using irrigation pipe:

Table 5. List of Tasks for 'Cultivate the plants' and 'The farmer fertilises using the irrigation pipe' scenarios.

Task	Methods
Cultivate the plants(Farmer, fertilizer, irrigation pipe)	{M1}
Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)	{M2}
Water(Farmer, plants)	{}//terminal task
Calculate amount(Farmer, minerals)	{}//terminal task
Pour(Farmer,mixture,irrigation pipe)	{}//terminal task
Activate(Farmer, irrigation pipe)	{}//terminal task
Pour(Farmer, water, irrigation pipe)	{}//terminal task

Table 6. List of methods of for 'Cultivate the plants' and 'The farmer fertilises using the irrigation pipe' scenarios.

Method: M1 **header**: Cultivate the plants(Farmer, fertiliser, irrigation pipe) **Control**: Water(Farmer, plants); Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals); **subtasks**: {Water, Fertilize using the irrigation pipe}
Method: M2 **header**: Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) **Control**: Calculate amount(Farmer, minerals); Pour(Farmer,mixture,irrigation pipe); Activate(Farmer, irrigation pipe); Pour (Farmer, water, irrigation pipe); **subtasks**: {Calculate amount, Pour, Activate, Pour}

- **subtask success**: Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)

Test case: The farmer fails to water plants:

- **subtask failure**: Water(Farmer, plants)

Test case: The farmer fails to fertilise using the irrigation pipe:

- **subtask failure**: Fertilize using the irrigation pipe (Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)

For the scenario 'The farmer fertilises using the irrigation pipe':

- **Test case**: The farmer fertilises using the irrigation pipe by calculating the amount of minerals, by pouring the mixture into the irrigation pipe, by activating the irrigation pipe and by pouring fresh water in the irrigation pipe:

- **subtask success**: Calculate amount(Farmer, minerals)
- **subtask success**: Pour(Farmer,mixture,irrigation pipe)
- **subtask success**: Activate(Farmer, irrigation pipe)
- **subtask success**: Pour(Farmer, water, irrigation pipe)
- **Test case**: The farmer fails to calculate the amount of minerals because:

failure of subtasks: Calculate amount(Farmer, minerals)

- **Test case**: The farmer fails to pour the mixture into the irrigation pipe:

failure of subtasks: Pour(Farmer,mixture,irrigation pipe)

- **Test case**: The farmer fails to activate the irrigation pipe:

failure of subtask: Activate(Farmer, irrigation pipe)

- **Test case**: The farmer fails to pour fresh water into irrigation pipe:

failure of subtask: Pour(Farmer, water, irrigation pipe)

3.6 Building a task/method model from scenario description. In this section, we briefly present how scenarios can be translated into task/method model. More details can be found in (Antonelli et al., 2018, 2020). The translation follows certain rules described below:

Rule 1 (Tasks Identification): *Each verb in the Episodes attribute (of the scenario model) is translated into a task in Task/Method model. Moreover, scenario title is also translated by a task .*

For example, if we apply rule 1 in the scenario 'Cultivate the plants', we obtain:

- Scenario: Cultivate the plant → Task: Cultivate the plants
- Episode: The farmer waters the plants → Task: Water

Rule 2 (Task's Parameters Identification): *In scenarios, each resource and each actor linked to an action is translated in task/method model into a parameter of the task corresponding to the linked action.*

In the scenario 'Cultivate the plants', rule 2 produces:

- Episode: The farmer waters the plants → Task: Water(Farmer, plants)

Rule 3 (Scenario's and Episode's method:) *Each way of realising an episode or a scenario is translated by a method task/method model.*

For example:

- Realization of Cultivate the plants → Method: M_1
- Realization of the episode: The farmer fertilises using irrigation pipe → *Method:* M_2

Rule 4 (Sequence of tasks): *Lines in 'Episodes' part of scenario described a sequence, thus they are translated by a sequence in control attribute of method.*

For example:

Episodes for the scenario 'Cultivate the plants':

The farmer waters the plants

The farmer fertilises using irrigation pipe

↓

Method: M1 Control

{

Water(Farmer, plants);

Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals);

}

Test cases generation The test cases correspond to all possible executions of tasks. Therefore, this step consists in computing all these executions and storing them in a data structure. Since task/method models are intrinsically hierarchical, this step uses a tree data structure to store all the executions. This structure is called Execution Tree (ET).

An ET contains two types of nodes:

Definition 3. *An etask node describes the execution of one task. It is composed of an execution status (success or failure) and a link to the description of the executed task (in the task/method model).*

Definition 4. *An emethod node represents an executed method. It owns an execution status (success or failure) and has access to the executed method in the task/method model.*

Figure 4 shows an ET for the task 'Cultivate the plants(Farmer, fertiliser, irrigation pipe)'. It contains the tree of all possible executions of the 'Cultivate the plants' task. In an ET, there is an alternation between *etask* nodes and *emethod* nodes. In Figure 4, *etasks* are represented by a box and *emethods* by an oval, an alternation between boxes and ovals can be seen. Failed *etasks* and *emethods* are displayed on a grey background and successful *etasks* and *emethods* on a white background. In this tree, five methods 'M3', 'M4', 'M5', 'M6' and 'M7' correspond to all possible executions of the task 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)'. The method 'M3' succeeded and the others failed. The *emethod* 'M5' failed because 'Calculate amount (Farmer, minerals)' succeeded but 'Pour(Farmer, mixture, irrigation pipe)' failed. An ET is generated by an execution engine.

The principle of the propagation of the execution status is: the terminal tasks succeed or fail, this status brings up the method that contains them. If the last subtask of a method fails, the method fails. If a task has at least one successful method, then the task has a success status.

The execution engine algorithm is provided in Algorithm 1. For the terminal task, two *emethods* are created one with the failure status and the other with the success status. For each method of non-terminal task, an *emethod* is created and the control

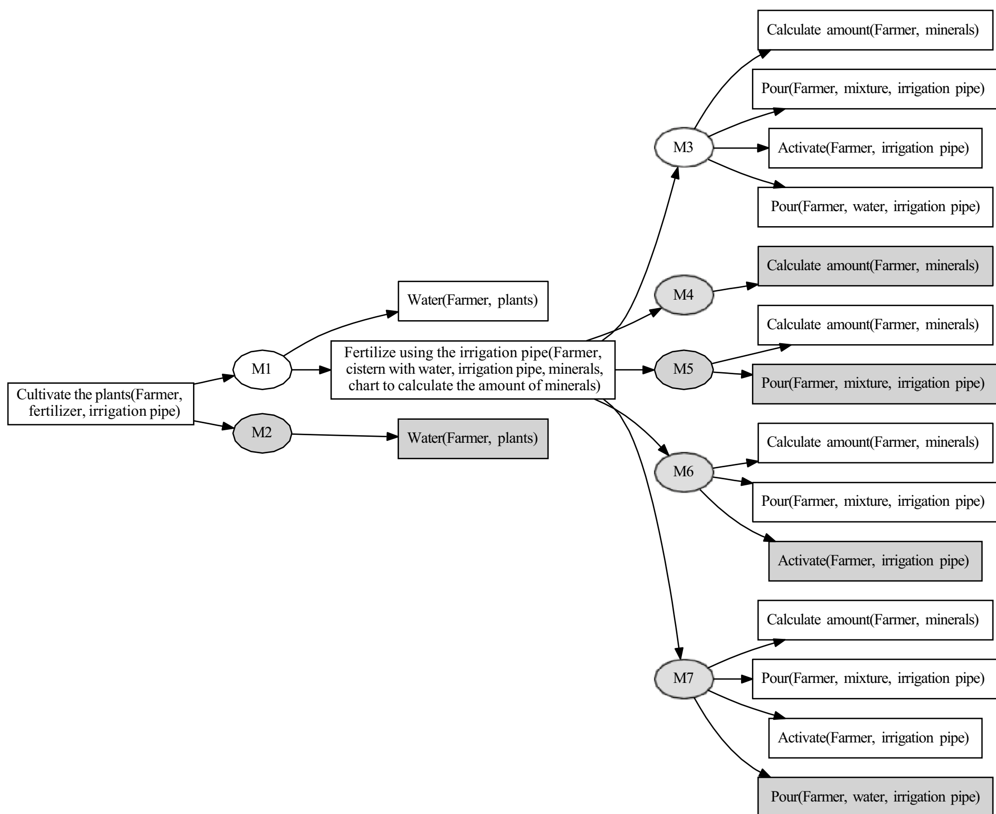

Figure 4. Execution tree for "Cultivate the plants" task.

attribute of the *emethod* is launched. The control attribute describes the execution order of the subtasks that are executed by recursively calling the execution engine on each of them (see Algorithm 1).

The ET presented in Figure 4 was built by this algorithm for the task 'Cultivate the plants(Farmer, fertiliser, irrigation pipe)'.

The test cases are extracted from the ET. In ET, one failure test case corresponds to one path from the root of the tree to a terminal task with the failed status. for example in Figure 4, the path 'Cultivate the plants' → 'Fertilize using the irrigation pipe' → 'Pour(Farmer, mixture, irrigation pipe' with failed status. This path can be read in the following way: 'Cultivate the plants' fails because 'Water' succeeds, but 'Fertilize using the irrigation pipe' fails because although 'Calculate amount' succeeds, 'Pour' fails. Similarly, a successful test case corresponds to a path starting from the root, consisting only of successful methods and ending with a terminal task.

The algorithm which generates all paths of failed test cases is presented in Algorithm 2. Basically, this algorithm starts from failed terminal tasks and goes up to the root. For the ET of Figure 4, this algorithm produced the following paths:

- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M1', 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)', 'M4', 'Calculate amount(Farmer, minerals)']
- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M1', 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)', 'M5', 'Pour(Farmer, mixture, irrigation pipe)']
- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M1', 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)', 'M6', 'Activate(Farmer, irrigation pipe)']
- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M1', 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)', 'M7', 'Wash(Farmer, water, irrigation pipe)']
- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M2', 'Water(Farmer, plants)'].

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 em*1 from *em* with the status *failure*
4: **generate** an *etask et*1 from the task *t* with *em*1 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 em*2 from *et* with the status *success*
10: **launch** the control attribute of *em*2
11: **end if**
12: **end for**
13: **end if**
14: **return** et

The generation of all test cases of success paths follows the same algorithm but selects only successful methods. For the ET of Figure 4, the success path is:

- ['Cultivate the plants(Farmer, fertiliser, irrigation pipe)', 'M1', 'Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals)', 'M3', 'Pour(Farmer, water, irrigation pipe)'].

Algorithm 2 Generation of Test Case paths 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 $p_{\lfloor\rfloor}$from the root of *ET* to *et*
5: T_Cases=T_Cases{*p*}
6: **end for**
7: **return** T_Cases

From the test case paths, we can generate a narrative expression using some natural language tools. For the example of Figure 4, we applied a very basic sentences generation and we obtained the following test cases in natural language:

- Cultivate the plants(Farmer, fertiliser, irrigation pipe) succeeds because Water (Farmer, plants) succeeds, and Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) succeeds because Calculate amount(Farmer, minerals) succeeds, Pour(Farmer, mixture, irrigation pipe) succeeds, Activate(Farmer, irrigation pipe) succeeds, and Pour (Farmer, water, irrigation pipe) succeeds.
- Cultivate the plants(Farmer, fertiliser, irrigation pipe) fails because Water(Farmer, plants) succeeds, but Fertilise using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) fails because Calculate amount(Farmer, minerals) fails.
- Cultivate the plants(Farmer, fertiliser, irrigation pipe) fails because Water(Farmer, plants) succeeds, but Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) fails because Calculate amount(Farmer, minerals) succeeds, but Pour(Farmer, mixture, irrigation pipe) fails.

- Cultivate the plants(Farmer, fertiliser, irrigation pipe) fails because Water(Farmer, plants) succeeds, but Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) fails because Calculate amount(Farmer, minerals) succeeds, Pour(Farmer, mixture, irrigation pipe) succeeds, but Activate(Farmer, irrigation pipe) fails.

- Cultivate the plants(Farmer, fertiliser, irrigation pipe) fails because Water(Farmer, plants) succeeds, but Fertilize using the irrigation pipe(Farmer, cistern with water, irrigation pipe, minerals, chart to calculate the amount of minerals) fails because Calculate amount(Farmer, minerals) succeeds, Pour(Farmer, mixture, irrigation pipe) succeeds, Activate(Farmer, irrigation pipe) succeeds, but Wash(Farmer, water, irrigation pipe) fails.

- Cultivate the plants(Farmer, fertiliser, irrigation pipe) fails because Water(Farmer, plants) fails.

4 Related works

The main activity for test design is finding defects instead of solving them. Testing is a very complex task included in a more generic process of software development life cycle: architecture, design, code, etc. Some researchers focus their contributions on the cognitive processes of software testers in order to contribute to the field (Enoiu et al., 2020).

For some application domains, testing is very critical. For example, software testing in automated vehicles is crucial to launch safe and reliable vehicles (Masuda, 2017). This application domain is characterised by an extremely large space of test input and a high cost of test executions. The first one is our main concern: a large space of test input. Thus, our proposed approach uses Scenarios to analyse their episodes in a combinatorial way to obtain the whole space of alternatives. It can be categorised 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.

Ramler and Klammer (Ramler & Klammer, 2019) introduce scenarios as models and use them to increase the coverage and reduce the effort of test design. They report their experience applying model-based testing for several real-world industrial projects where they were able to minimise the risks and reduce the effort in testing.

A literature review of test design approaches is presented by Dos Santos et al. (Dos Santos et al., 2018). Their findings show that there is no existing approach that incorporates a supportive tool utilising natural language text descriptions, in particular, a behaviour-driven technique using scenarios. These are the key elements of our approach. Natural language is a key element in obtaining descriptions directly from end users, those who will use the application. Another issue is that it is mandatory that the end-users accept it. Moreover, Scenarios describing the behaviour of the application domain constitute an excellent approach since scenarios tell stories that are easily understandable. Other works deriving Uses Cases to Users Acceptance Tests (UAT) (Bystrický & Vranić, 2017; C. Y. Hsieh et al., 2013) or acceptance criteria described in the form of Given-When-Then (Pandit et al., 2016) are also frequently used. All the aforementioned approaches require an effort to build these models. Moreover in agile methodologies, that do not use artefacts such as UAT (Jeeva Padmini et al., 2016).

Svensson and Regnell (Svensson & Regnell, 2015) state that automating testing is a shared concern in software engineering. Testing software generally requires a lot of effort from programmers. They should imagine all the possible errors that the end-users could make. They also have to work with a lot of rigour 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 and Elberzhager (Garousi & 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 applications with two main parts: (i) test case generation and (ii) test case execution. It is important to remark that both proposals include one first step (separated from the others) related to the design of the test cases.

Monpratarnchai et al. (Monpratarnchai et al., 2013) propose an approach to generate test cases described in JUnit from Java source code. A combined approach is proposed by Lipka et al. (Lipka et al., 2015). They derive test cases from requirements and source code. They consider narrative requirements enriched with annotations to connect the specification to the source code. An interesting technique based on Use Cases is proposed by Khamaisehand Xu (Khamaiseh & Xu, 2017). They determine misuse cases to test vulnerabilities. Philip et al. (Philip et al., 2017) approach is similar since they analyse a model with safety requirements to generate fault trees representing functional hazards. Then, test cases for the validation of the mitigation of hazards are generated automatically from the model.

It exists many other proposals based on requirements, some others on conditions, restrictions, or states. These elements could be captured using Use Cases, formal languages, state machines, or workflow diagrams that are the input of the approach developed by Chatterjee and Johari (Chatterjee & Johari, 2010). They propose an approach to derive test cases from Use Cases, as well as (Bouquet et al., 2008). Although they analyse the alternatives in the flow of actions, the preconditions stated in the Use Cases, they finally rely on a state machine. On the contrary, our approach relies on combining all the possible actions (and their result).

Pandit et al. (Pandit et al., 2016) propose an approach to design User Acceptance Tests. This approach is similar to our approach. Nevertheless, they base their proposal on acceptance criteria written in the form of a Given-When-Then template. They also rely on states that are arranged in a dependency graph. Lei and Wang (Lei & Wang, 2016) propose a framework to analyse testing constraints in requirements, that is, another way of considering states. Huaikou et al. (Huaikou & Ling, 2000) analyse specifications, in particular, the prior and posterior state of every operation to generate test cases.

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

Hsieh et al. (C. Hsieh et al., 2013) propose an approach to analyse the steps of the Use Cases to determine all the alternatives to design tests in order to cover all the possibilities. Other approaches are based on the external relations of the Use Cases. This is the case of

Lizhe Chen and Qiang Li (Chen & Li, 2010). who 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 explore a tree to obtain all the alternatives. Boucher Mussbacher (Boucher & 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 to generate test cases from use cases with a specific definition of control flow, input, and output. Vieira et al. (Vieira et al., 2006) propose a similar approach using annotated UML Activities Diagrams.

Entin et al. (Entin et al., 2009) propose to focus on obtaining tests independent from the platform. They obtain very general User Acceptance Tests as the one obtained by our approach. Takagi and Noda (Takagi & Noda, 2016) describe a strategy to develop a graph about the sequence of test case execution related to hardware testing, that is very detailed and specific situations in contrast with the essence of the Scenarios. Hussain et al. (Hussain et al., 2015) provide design tests considering dependencies between Scenarios. Nomura et al. (Nomura et al., 2014) model business context in a matrix representing the dependencies between the business process. The tests are then designed from the perspective of profiles in order to cover different situations. Sarmiento et al.

(Sarmiento et al., 2016) propose a similar approach using scenarios.

Summing up, our proposed approach satisfies some characteristics that, to our better knowledge, they are not satisfied by another proposal. That is, our proposed approach deals with a large set of scenarios that have behaviour-driven text descriptions, and our proposed approach also provides a supporting tool that deals with natural language.

5 Conclusions

This paper proposed an approach to design test cases. It consists in analysing natural language artefacts using them very early in the software development life cycle instead of using artefacts of the design step as usually done. Thus, this approach makes it possible to contrast the software application with the initial requirements. The approach relies on using and mixing several tools and techniques. It uses natural language processing tools in order to cope with the description of the scenarios to provide a consistent and coherent description regarding the variety of stakeholders involved and their differences in the descriptions. It also uses the task/method model to deal with the task of systematically analysing all the situations that is relevant to tests. And this is a crucial feature of the proposed approach. Although some approaches provided a very detailed sequence of steps to perform unit or integration testing, our proposed approach provides a guideline for user acceptance testing in order to assess the correctness of the software application. The main advantage of our approach is to propose a systematic way to test software thanks to the Task/method paradigm. Nevertheless, a systematic approach is very time consuming, so we propose to optimise this approach using a Natural Language Processing (NLP) paradigm in order to automatically generate software tests. We plan to continue this research in three different ways. Firstly, our proposed approach considers that scenarios can be organised in one big tree with scenarios in the different levels.

Although there is no limitation to the number of levels in the tree, we consider that only one tree is generated. That is, we consider that there is one root scenario that includes episodes that should be related to scenarios, that in turn, are related to other scenarios and finally, all the scenarios should be stuck to this tree. It makes sense, since a software application should integrate the whole functionality, so, all the scenarios should be integrated into the tree. Nevertheless, our proposed approach does not provide an integrated set of test cases if there is not only one scenario as a root. This could happen in different situations. For example, if the set of scenarios is not complete and some scenarios can not be related to the tree. Or if the description of the scenarios is too complex, and the natural language processing techniques provided in the approach are not able to link the scenarios. We believe that in that case, the proposed approach should raise some warning about the scenarios that can not be related to the consolidated tree of scenarios to rewrite them or to add more scenarios. This situation could be even more complex since one node of the tree for example, could also be linked to different scenarios at the same time. For example, if an episode makes reference to simple"fertilises" and there are two strategies to do that: (i) 'Fertilises using the irritation pipe' and (ii)'fertilises using the backpack', the node"fertilises" should be related to both different ways of fertilising. Regarding the composition of scenarios, we also plan to explore the integration of the Language Extended Lexicon (LEL) (Do Prado Leite et al., 2000), a technique to capture and model the glossary of the application domain. This LEL glossary is coupled in many proposed methods to the scenarios technique used in this proposed approach. Thus, we believe that their use will be beneficial. Moreover, we believe that we can take advantage of some attributes of the scenarios not used: restrictions and exceptions in order to relate scenarios. For example, regarding the fertilisation example, the exception attribute of the scenario 'Fertilise using the irrigation pipe' could state that 'The cistern does not have enough water to activate the irrigation pipe' and the solution to this exception could be: 'Fertilise using the spraying backpack'. Thus, the link between the scenarios is explicitly mentioned. We think that the two incorporations, the LEL glossary and the attributes of restrictions and exception will demand more effort of the practitioners since they need to describe more information, but the results will be better. Secondly, our proposed approach only considers functional requirements, that is, activities described in the scenarios are woven by the proposed approach that provides all the combinations that should be tested to assure the functionality. Nevertheless, non-functional requirements are not considered. They are much more complex to deal with, since non-functional requirements are specified in one sentence of a scenario, but that characteristic could be scattered in all the software systems, so this should be tested in many different places. For example, let's consider the security non-functional requirements. If the application should be secure, one root scenario can include this characteristic and all the children of the root scenario should ensure security. Thus, dealing with non-functional requirements requires spreading them around the necessary scenarios. Thirdly, we plan to perform more validations. We plan to perform a case study in order to assess the applicability and usability of the proposed approach. And we also plan to perform a controlled experiment in order to compare the effectiveness of our proposed approach against some other approaches. It is important to note that we voluntarily do not use all information available in scenarios and all representation capabilities of the Task/method paradigm in the translated Task/Method models. Our strategy is to

complexify the Task/Method as little as possible, and to add representation elements (fields for example) only in order to be able to generate interesting test cases. Regarding the semantic representation, we will study the combination with other types of ontologies defined in the elicitation phase, for example a combination with the Requirements Journey ontology defined by (Lane et al., 2016) which includes concepts as User, Event, or Timeline.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

The work was supported by the This paper is partially supported by funding provided by the STIC AmSud program, Project 22STIC-01 This paper is partially supported by funding provided by the STIC AmSud program, Project 22STIC-01.

ORCID

Leandro Antonelli http://orcid.org/0000-0003-1388-0337

References

Abeysiriwardana, P.C., Jayasinghe-Mudalige, U.K., & Seneviratne, G. (2022). Probing into the concept of 'research for society' to utilize as a strategy to synergize flexibility of a research institute working on eco-friendly commercial agriculture. *All Life*, *15*(1), 220–233. https://doi.org/10.1080/26895293.2022.2038280

Adla, A., Soubie, J.L., & Zarate, P. (2007). A co-operative intelligent decision support system for boilers combustion management based on a distributed architecture. *Journal of Decision Systems*, *16*(2), 241–263. https://doi.org/10.3166/jds.16.241-263

Alexander, I., & Maiden, N. (2004). Scenarios, stories, and use cases: The modern basis for system development. *Computing Control Engineering Journal*, *15*(5), 24–29. https://doi.org/10.1049/cce:20040505

Antonelli, L., Camilleri, G., Grigera, J., Hozikian, M., Sauvage, C., & ZARATÉ, P. (2018, May). A modelling approach to generating user acceptance tests. In: F. Dargam, P. Delias, I. Linden, & B. Mareschal (Eds.) 4th International Conference on Decision Support Systems Technologies (ICDSST 2018). Decision Support Systems VIII: Sustainable Data-Driven and Evidence-Based Decision Support, 313. Springer, Heraklion, Greece, https://hal.archives-ouvertes.fr/hal-02289948

Antonelli, L., Hozikian, M., Camilleri, G., Fernandez, A., Grigera, J., Torres, D., & Zaraté, P. (2020). Wiki support for automated definition of software test cases. *Kybernetes*, *49*(4), 1305–1324. https://doi.org/10.1108/K-10-2018-0548

Boucher, M., & Mussbacher, G. (2017). Transforming workflow models into automated end-to-end acceptance test cases. 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. p. 45–48. AST '08, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1370042.1370052

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, Kanyakumari, India. vol. 1, pp. 178–182.

Bystrický, M., & Vranić, 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. https://doi.org/10.15439/2017F343

Camilleri, G., Soubie, J.L., & Zalaket, J. 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. pp. 45–52. Springer (2003 september), http://www.springerlink.com/index/CMXAFDNLQCAJDKVX%20, pages de la publication : 45–52, partI.

Chatterjee, R., & Johari, K. (2010, October). A prolific approach for automated generation of test cases from informal requirements. *Software Engineering Notes*, *35*(5), 1–11. https://doi.org/10.1145/1838687.1838702

Chen, L., & Li, Q. (2010). Automated test case generation from use case: A model based approach. In: 2010 3rd International Conference on Computer Science and Information Technology, Chengdu, China. vol. 1, pp. 372–377.

Do Prado Leite, J.C.S., Hadad, G.D.S., Doorn, J.H., & Kaplan, G.N. (2000). A scenario construction process. *Requirements Engineering*, *5*, 38–61. https://doi.org/10.1007/PL00010342

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), Gothenburg, Sweden. pp. 418–419.

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. https://doi.org/10.1109/QRS-C51114.2020.00053

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 Cumberland Lodge, Windsor, United Kingdom. pp. 179–183.

Forsberg, K., Mooz, H. (1991). The relationship of system engineering to the project cycle. In: Proceedings of the First Annual Symposium of National Council on System Engineering, Chattanooga, TN, USA. pp. 57–65.

Garousi, V., & Elberzhager, F. (2017). Test automation: Not just for test execution. *IEEE Software*, *34*(2), 90–96. https://doi.org/10.1109/MS.2017.34

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), San Francisco, CA, USA. 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. https://doi.org/10.1109/IWAST.2013.6595797

Huaikou, M., & Ling, L. (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, Tokyo, Japan. pp. 164–171.

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) Peshawar, Pakistan. pp. 1–6.

ISO/IEC/IEEE. (2010) Systems and software engineering – vocabulary. ISO/IEC/IEEE, 24765:2010 edn.

ISO/IEC/IEEE. (2014) IEEE draft international standard for software and systems engineering–software testing–part 4: Test techniques. Tech. rep., IEEE.

Jackson, M. (1995). The world and the machine. In: 1995 17th International Conference on Software Engineering, Seattle, Washington, USA. 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. https://doi.org/10.1109/MERCon.2016.7480122

Karlsson, L., Dahlstedt, Å.G., Regnell, B., Natt Och Dag, J., & Persson, A. (2007). Requirements engineering challenges in market-driven software development – an interview study with practitioners. *Information and Software Technology*, *49*(6), 588–604. https://www.sciencedirect.com/science/article/pii/S0950584907000183. qualitative Software Engineering Research.

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), Hangzhou, China. pp. 534–541.

Lane, S., O'Raghallaigh, P., & Sammon, D. (2016). Requirements gathering: The journey. *Journal of Decision Systems*, *25*(sup1), 302–312. publisher: Taylor & Francis. https://doi.org/10.1080/12460125.2016.1187390

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

Lipka, R., Potuák, T., Brada, P., Hnetynka, P., & Vinárek, 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, Madeira, Portugal. pp. 241–244.

Masuda, S. 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. (2017). https://doi.org/10.1109/ICSTW.2017.55

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), Ratchathewi, Bangkok, Thailand. vol. 2, pp. 93–98.

Niazi, A.R., & Ghafoor, A. (2021). Different ways to exploit mushrooms: A review. *All Life*, *14*(1), 450–460. https://doi.org/10.1080/26895293.2021.1919570

Nogueira, S., Sampaio, A., & Mota, A. (2014). Test generation from state based use case models. *Formal Aspects of Computing*, *26*(3), 441–490. https://doi.org/10.1007/s00165-012-0258-z

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, Jeju, South Korea. vol. 1, pp. 3–10.

Pandit, P., Tahiliani, S., & Sharma, M. (2016). Distributed agile: Component-based user acceptance testing. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN), Indore, Madhya Pradesh, India. 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), St. Petersburg, Florida, USA. pp. 1–10.

Prince, I.A., Adnan, M.A., Rifat, R.I., Mostafiz, M.S., & Rahman, S.I. (2022). Iot based monitoring framework for a novel hydroponic farm. In: 2022 IEEE Region 10 Symposium (TENSYMP). pp. 1–4. https://doi.org/10.1109/TENSYMP54529.2022.9864365

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. https://doi.org/10.1109/QRS-C.2019.00096

Saah, K.J.A., Kaba, J.S., & Abunyewa, A.A. (2022). Inorganic nitrogen fertilizer, biochar particle size and rate of application on lettuce (Lactuca *sativa* L.) nitrogen use and yield. *All Life*, *15*(1), 624–635. https://doi.org/10.1080/26895293.2022.2080282

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 . http://www.sciencedirect.com/science/article/pii/S1571066116301153

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. https://doi.org/10.7551/mitpress/4073.001.0001

Shruthi, G., Selva Kumari, B., Rani, R.P., & Preyadharan, R. (2017). A-real time smart sprinkler irrigation control system. In: 2017 IEEE International Conference on Electrical, Instrumentation and Communication Engineering (ICEICE). pp. 1–5. https://doi.org/10.1109/ICEICE.2017.8191943

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, San Francisco, CA, USA USA. pp. 274–279.

Svensson, R.B., Regnell, B. (2015). Aligning quality requirements and test results with quper's road-map view for improved high-level decision-making. In: 2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing, Florence, Italy. 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), Tokyo, Japan. 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. https://doi.org/10.1016/S0957-4174(98)00065-7

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. p. 9–14. AST '06, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1138929.1138932

Yang-Ren, W., & Zhi-Wei, Z. (2016). Research of tomato economical irrigation schedule with drip irrigation under mulch in greenhouse. In: 2016 Fifth International Conference on Agro-Geoinformatics (Agro-Geoinformatics). pp. 1–5. https://doi.org/10.1109/Agro-Geoinformatics.2016.7577636

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