International Journal of Electrical and Computer Engineering (IJECE)
Vol. 13, No. 1, February 2023, pp. 809~822
ISSN: 2088-8708, DOI: 10.11591/ijece.v13i1.pp809-822

Automatic generation of business process models from user stories

Samia Nasiri, Amina Adadi, Mohammed Lahmer
LMMI Laboratory of ENSAM, ISIC Research Team of ESTM, Moulay Ismail University, Meknes, Morocco

Article Info

Article history:

Received Feb 16, 2022
Revised Jul 19, 2022
Accepted Aug 19, 2022

Keywords:

Business process modeling
UML activity diagram
Natural language processing
Requirements engineering
User stories

ABSTRACT

In this paper, we propose an automated approach to extract business process models from requirements, which are presented as user stories. In agile software development, the user story is a simple description of the functionality of the software. It is presented from the user's point of view and is written in natural language. Acceptance criteria are a list of specifications on how a new software feature is expected to operate. Our approach analyzes a set of acceptance criteria accompanying the user story, in order, first, to automatically generate the components of the business model, and then to produce the business model as an activity diagram which is a unified modeling language (UML) behavioral diagram. We start with the use of natural language processing (NLP) techniques to extract the elements necessary to define the rules for retrieving artifacts from the business model. These rules are then developed in Prolog language and imported into Python code. The proposed approach was evaluated on a set of use cases using different performance measures. The results indicate that our method is capable of generating correct and accurate process models.

Corresponding Author:

Samia Nasiri
LMMI Laboratory of ENSAM, ISIC Research Team of ESTM, Moulay Ismail University
Meknes, Morocco
Email: nasiri.samia@gmail.com

1. INTRODUCTION

Requirements modeling is one of the earliest phases of software development. Its main objective is to understand and support the expected needs of future software development. Requirements modeling allows analysts to determine the real needs of stakeholders and to communicate with them in a language they understand, such as models instead of complex texts [1]. This analysis is based on requirements that formulate future software needs presented in natural language, described in user stories, and detailed in acceptance criteria, precisely in agile projects. Like other models resulting from this design phase, the behavior model is based on the collected requirements [2]. Acceptance criteria also called "definition of done", represent requirements that must be performed by a developer in agile software development. In the software development process, the communication between the team developers is crucial, the user story which is a requirements format is not sufficient to describe the tasks and events related to an action in the business process [3]. For this, the acceptance criteria are used to solve the incompleteness problems that can be caused in the user story. To define the acceptance criteria, natural language is used in different ways such as a list or a given-when-then (GWT) model. Many approaches in the literature cover either unified modeling language (UML) diagrams generation from textual requirements [4]–[12] or test case generation from acceptance criteria in agile projects [13]–[15], Yet, few approaches support the generation of business process models from textual requirements, either unstructured requirements [16], [17] or agile requirements

[18]. In [17], the UML activity diagram is generated automatically from natural language requirements. The limitation of this work is that some controls like decision nodes and forks are not automated. In [18], the business process model is built automatically from unrestricted requirements, but by manipulating the agile requirements, the authors have only managed to generate the operations of the model, and their model is still incomplete. Moreover, they only process the user stories without taking into account the acceptance criteria. In our work, we address the aforementioned research gaps and present an approach to minimize the manual effort required to derive a business process model from acceptance criteria written in GWT format using natural language processing (NLP) techniques.

In this article, we propose our method of automatically generating a business process model, which in our case is an activity diagram. This model is generated from a set of user stories grouped and analyzed with acceptance criteria. Our approach starts with the use of NLP techniques to extract the elements necessary to define the rules for retrieving artifacts from the business model. these artifacts are activities and actions describing the business process. The defined rules are developed into Prolog language and imported in Python code to complete the production of the activity diagram as a PNG image using PlantUML.

The structure of this document is as follows. This section presents the requirements modeling, and our proposed approach. Furthermore, we detail our method in section 2. Then, in section 3, we expose and analyze a UML activity diagram generated from a set of acceptance criteria. Thereafter, the performance of our approach is evaluated and discussed. Finally, the conclusion is presented in section 4.

2. METHOD

2.1. Related work

In this subsection, we present a literature review related to our approach. In [13] SPECMATE is developed based on acceptance criteria in order to generate automatically case tests using NLP techniques. The core of their work consists of the use of a dependency parser provided by the NLP tool. The fragmentation of the dependency parser tree into causes and effects is carried out. To cover a number of different cause-and-effect relationships, the authors created a set of patterns in English and German. They developed an algorithm that navigates a dependency tree from its root and applies the patterns to the several nodes in the tree. Pandit and Tahiliani [14] have developed a framework called AgileUAT, their approach aims to generate a test acceptance in a test use case diagram from user stories and acceptance criteria, AgileUAT generates the acceptance test cases in natural language. The test cases that are obtained from the acceptance criteria are stored in an Excel sheet. Sibal *et al.* [15] propose a technique for prioritizing user story acceptance tests and for determining the critical acceptance tests. Their approach is based on acceptance criteria. To achieve their goal meta-heuristic algorithms are used, i.e., genetic algorithm and cuckoo search algorithm. The UML activity diagram is drawn and converted into a control flow graph. The authors in [16] proposed an approach allowing the generation of the business process modeling notation (BPMN) model from the textual requirement. The use of NLP was essential to analyze the specifications. In [18] iterative approach of models extraction from the requirements (iMER) is developed to automatically generate the entity relationships model and business process model from text requirements and user stories. Their approach is made iterative in order to extract the component of each model.

In [19] a set of rules is defined by the ATLAS transformation language (ATL) for the semi-automatic transformation from the computing independent model (CIM) level to the platform independent model (PIM) level in model driven architecture (MDA). The CIM level is presented by a BPMN model, and the PIM level contains three UML diagrams, namely the use case diagram, the state diagram, and the class diagram. The approach proposed in [20] allows for generating a BPMN model with a decision table schema integrated with the rules. This model is automatically generated from the model of the relations between attributes. In [21] the generation of BPMN is performed from a workflow log. The development of the process model generator is carried out with the help of Python with the pm4py library which provides the implementation of alpha and inductive miner algorithms and allows the presentation of the discovered processes in the form of Petri nets, as with their conversion into a BPMN diagram in XML and graphic format. Bouzidi *et al.* [22] adopt an MDA approach named BPMN2US. This method allows generating the use case diagram with the textual description of use cases. In order to generate a use case diagram from the BPMN model, a set of transformation rules is defined using the ATL language. In addition, they have developed templates using Acceleo to provide the textual description of the use cases. In [23] the approach of the authors is based on a business process model to generate the natural language requirements. Their approach is semi-automatic and consists of two steps, namely the preparation phase and the generation phase. In the preparation phase, the analysis of the business process is carried out to detect the automated activities. then a requirement model is built for each activity. In the generation phase, the requirements model represents the input of this phase to generate the requirements document, this generation phase follows three steps: first, sentences are generated, then they are refined, and finally they are organized into a document that represents the requested requirements. In [24], a tool named MARITACA is developed by using NLP techniques to generate the state machine model from the use case diagram. A set of rules has been applied to the word sequences for detecting states, transitions, and triggers. In [25] an automated approach is proposed to generate use case scenarios from user stories. The generated robustness diagram includes the behavioral scenarios. Spacy NLP tool is used for analyzing the user stories and extracting actors, boundaries, controls, and entities. In [26], using process mining algorithm, the authors were able to generate the BPMN model in a PNG format and SVS format. This generation is the result of the analysis of descriptions specified with a textual domain specific language (DSL). Their approach uses textual DSL for process modeling.

As can be seen, many approaches for user acceptance testing have been developed so far. They can be classified into two groups: i) requirements-based and ii) business process-based. Our approach follows the requirements-based process in which user stories and acceptance criteria form the basis of our automation. Our approach aims at automatically generating activity diagrams from a set of acceptance criteria accompanying a user story.

2.2. Proposed approach

In this subsection we present our approach for extracting artifacts necessary to generate the activity diagram, our implementation was developed in Python language. Prolog language is used to define these extraction components. As a first step, we implemented our system using the Spyder development environment that is included in Anaconda, the Python distribution. In this tool, we created Python files in which we analyze user stories with acceptance criteria using the Stanford coreNLP tool for preparing elements to be used in Prolog rules. The defined rules allowed us to detect the activities and symbols composing the activity diagram. The Prolog file was created using the SWI-Prolog tool. From the Python files, we access the Prolog rule file using the PySwip API, this library allows us to query SWI-Prolog in Python programs. In the last step, we automatically grouped the components of the UML diagram into a text file that can be interpreted by the PlantUml API. This application programming interface (API) is implemented in Python language to generate the desired diagram as an image. Figure 1 shows the architecture of our proposed approach.

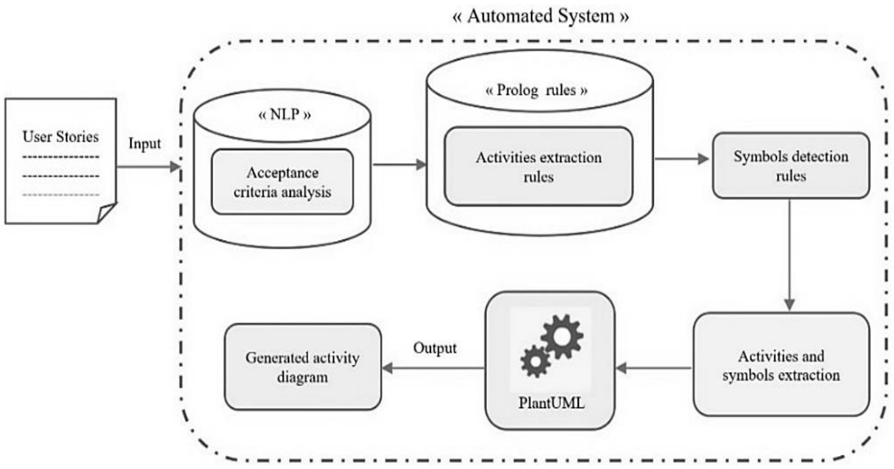

« Automated System »
User Stories
Input
« NLP »
Acceptance criteria analysis
« Prolog rules »
Activities extraction rules
Symbols detection rules
Activities and symbols extraction
PlantUML
Output
Generated activity diagram

Figure 1. Architecture of our proposed approach

2.2.1. User stories and acceptance criteria templates

In agile software development, requirements are expressed in documents called user stories. These latter are an effective and efficient way to describe the user's needs. A user story often uses the following format type: *As* <Role>, *I want to* <Action>, *so that* <Benefit>.

In Agile software development, every team uses acceptance criteria because they can be crucial to a successful implementation. Acceptance criteria are requirements that consist of a list of details on how a new software feature should operate. The input file in our approach is detailed in Figure 2.

Several teams using the agile methodology prefer the scenario-oriented type of acceptance criteria. This approach provides requirements, and it was inherited from behavior-driven development (BDD). A most used approach to writing acceptance criteria is the "*Given/When/Then*" scenario-oriented approach, its template is suitable for this format:

Given [some context], When [action] is carried out, Then [result] happens

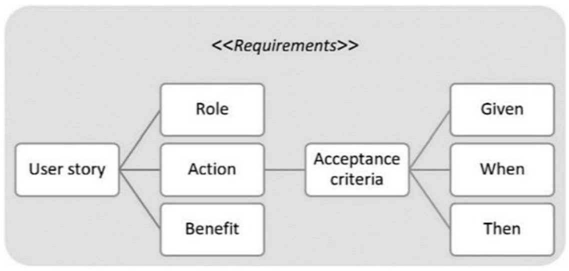


Figure 2. Structure of requirements in our proposal approach

2.2.2. Natural language processing

The user story and acceptance criteria are written in natural language. Thus, we have used an NLP tool for their preprocessing. This tool is called Stanford CoreNLP. Our approach starts by browsing a set of acceptance criteria to apply a coreference, which allows in our case to replace a pronoun with the subject. Stanford CoreNLP provides coreference resolution, so in our case, this functionality is implemented in the Python language. Figure 3 depicts the steps followed in the preprocessing stage of our extraction process. Figure 4 shows an example of applying the coreference resolution.

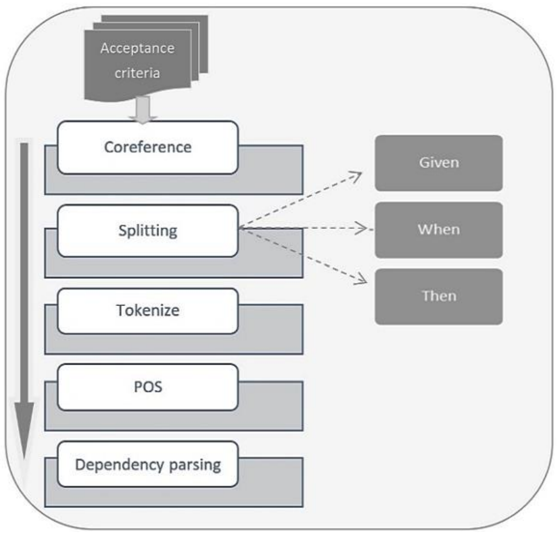
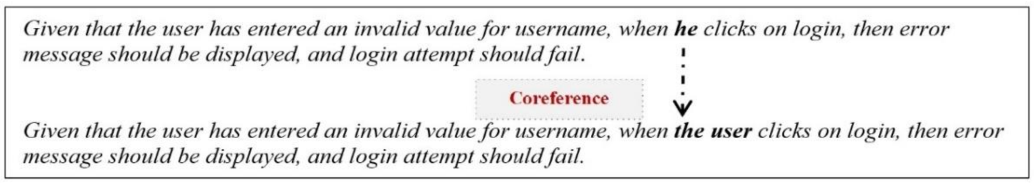


Figure 3. Steps of preprocessing the agile requirements


*Given that the user has entered an invalid value for username, when **he** clicks on login, then error message should be displayed, and login attempt should fail.*

Coreference

*Given that the user has entered an invalid value for username, when **the user** clicks on login, then error message should be displayed, and login attempt should fail.*

Figure 4. Example of applying coreference resolution to a user story

After applying the coreference for each acceptance criteria, we split it into three parts: "*given*", "*when*" and "*then*" parts. Then comes the tokenization step which allows splitting the sentence into several tokens. After tokenization, and we need to determine if the token is a noun or a verb. For this reason, the use of part of speech (POS) was necessary. In our case, the use of POS was not enough, so we use dependency parsing which provides a typed dependency that helps us a lot in the process of defining the artifact extraction rules. These typed dependencies represent grammatical relationships between the words of a sentence to extract textual relations. It is denoted by a triple relation between two pairs of words. Figure 5 shows a user story with its typed dependencies.

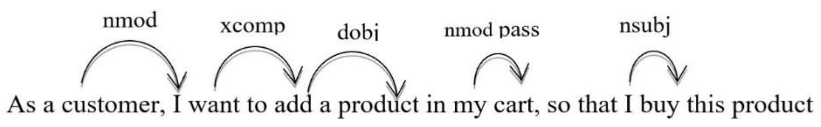

nmod
xcomp
dobj
nmod pass
nsubj
As a customer, I want to add a product in my cart, so that I buy this product

Figure 5. Example of a user story with its typed dependencies

2.2.3. Artifact extraction algorithm

To extract the components of the activity diagram from a set of acceptance criteria, we followed the steps described in the proposed algorithm. As an input, the algorithm takes a set of acceptance criteria (AC) and empty sets of activities *In* and *Out*, and actions A. Then *In*, A and *Out* are filled in by applying extraction rules. The loop in line 2 consists of parsing through each acceptance criteria. In line 3, coreference is applied to replace pronouns in the acceptance criteria. In lines 4-6, we divide the acceptance criteria into three parts to analyze them. In lines 7-9, the part of speech and tokenization are applied to classify each word in each part of the acceptance criteria. Then, typed dependencies are detected. In lines 10-12, the activities and actions are extracted. It should be noted that the same rules are applied in the three parts of the GWT, if a rule is applied in the second part, it is considered as an action or a condition and not an activity. However, the first and the third part are activities.

Algorithm. Artifact extraction
```
1: Procedure (Acceptance_criteria AC, activity IN, action A, activity Out)
2:   for each c in AC
3:   input=coreference(c)          -- coreference resolution of Stanford coreNLP
4:   g=extractGiven(input)
5:   w=extractWhen(input)
6:   t=extractThen(input)
7:   POS(g), POS(w), POS(t)
8:   word_tokenize(g), word_tokenize(w), word_tokenize(t)
9:   dependency_parse(g), dependency_parse(w), dependency_parse(t)
10:  IN=extract_inputActivity(g)
11:  A=extract_action(w)
12:  Out=extract_OutputActivity(t)
```

2.2.4. Rules for extracting the activity diagram components

In order to automatically generate an activity diagram from several acceptance criteria, it was necessary to extract the components of this diagram, namely the activities, the action, and the transition as well as the decision nodes. To do this, we used Prolog language to define rules for extracting artifacts. In this section, we present the rules for extracting the components of the activity diagram. These rules are based on the POS and on the type of dependencies offered by Stanford CoreNLP tool. Table 1 depicts the rules used to detect the activities and conditions that constitute the activity diagram.

The rules are developed in Prolog language in this form: activity (*X*, *Y*, *Z*). *X* is the action, *Y* is the performer of the action, and *Z* is the object of the action. The rules represent the combination of typed dependencies, where each dependency presents a relationship between two words. These words denote *X* or *Y* or *Z*. Our tool gathers these words to form the desired activities with this form: *X_Y_Z*.

In the majority of the rules defined, we have three or four variables that, when grouped together, become part of the activity diagram. Sometimes it is not enough to extract *X*, *Y*, and *Z* to present activity because this leads to incomplete information. To solve this problem, we add a fourth variable that depends essentially on the structure of the sentence. Prolog rules which contain four variables *X*, *Y*, *Z*, and *W* detect either a preposition presented by *Y* or a description of *Z* presented by *W*. Our approach supports the verification of the verbs' negation presence via the typed dependency called "neg". In the case of a verb with a preposition, we extract them together (*X_Y*) and after the subject and object of the verb (*Z_W*). Moreover, our approach takes into account the processing of compound nouns, and also the inference of an activity thanks to the conjunction between two nouns where the first noun is retrieved as part of a generated task. Verb preposition detection is performed using these dependencies "Case, Advmod and Compound: Prt" in Prolog rules as shown in Table 1.

Given some acceptance criteria with rule applications as listed below:

- AC#1: *Given* that the user is logged in, *when* the user clicks on the avatar, *then* the system opens the account page.
- AC#2: *Given* that a user has left username or password fields empty, *when* a user clicks on login, *then* error should be displayed, and login attempt should fail.
- AC#3: *Given* the account is overdrawn and the card is valid, *when* the customer asks for cash, *then* ensure a rejection message is displayed and ensure cash is not dispensed and ensure the card is returned.

Table 2 illustrates Prolog rules with their applied examples.

Table 1. Rules for activities detection

	Typed dependency															
Rules	nsubj	nsubj pass	Csubj pass	mod	dobj	advmod	case	Compound: prt	xcomp	ccomp	advcl	conj	ccop	nummod	dep	acl:relcl
R1	x				x											
R2	x	x		x		x	x					x				
R3	x			x								x				
R4	x												x			
R5	x			x	x	x										
R6	x				x	x					x					
R7	x				x			x								
R8	x	x			x	x	x	x								
R9		x		x	x	x	x	x						x		
R10	x									x						x
R11	x			x			x								x	
R12					x	x					x					
R13	x								x							x
R14	x						x						x		x	
R15			x		x										x	
R16									x		x				x	
R17			x		x	x										
R18		x		x	x	x								x		
R19	x			x	x	x								x		
R20	x			x												
R21	x	x														
R22		x								x						

Table 2. Prolog rules with their applied examples

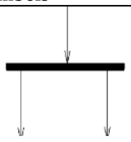
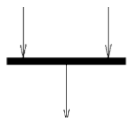
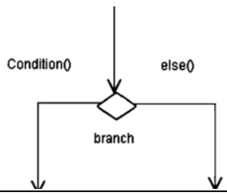
Acceptance Criteria	Typed dependencies		Applied rules in prolog language	Extracted activities
AC#1	- ('**nsubjpass**', 'logged', 'user')	R21	**Activity(X,Y,system):- ((nsubjpass(X,Y); nsubj(X,Y)),not(dobj(X,_)), not(nmod(X,_)),not(xcomp(X,_)).**	Logged_user_ system
	- ('**nsubj**', 'clicks', 'user'), ('**case**', 'avatar', 'on'), ('**nmod**', 'clicks', 'avatar')	R11	**Activity(X,Y,W,Z):- nsubj(X,Y),nmod(X,Z), case(Z,W).**	Clicks_user_on_ avatar
	- ('**nsubj**', 'opens', 'system'), ('**dobj**', 'opens', 'page')	R1	**Activity(X,Y,Z):- nsubj(X,Y), nmod(X,Z).**	Opens_system_account _page
AC#2	- ('**xcomp**', 'left', 'username'), ('**dep**', 'username', 'fields'), ('**dep**', 'username', 'empty	R16	**Activity(X,Y,Z) :- xcomp(X,Y),dep(Y,Z).**	Left_username_field_ empty
	- ('**nsubj**', 'clicks', 'user'), ('**case**', 'login', 'on'), ('**nmod**', 'clicks', 'login')	R11	**Activity(X,Y,W,Z):- nsubj(X,Y),nmod(X,Z), case(Z,W).**	Clicks_user_on_login
	- ('**nsubjpass**', 'displayed', 'error'), ('**nsubj**', 'fail', 'attempt')	R21	**Activity(X,Y,system):- ((nsubjpass(X,Y);nsubj(X,Y)), not(dobj(X,_)), not(nmod(X,_)), not(xcomp(X,_)).**	Displayed_error_ system Fail_login_attempt_ system
AC#3	- ('**nsubj**', 'valid', 'card'), ('**cop**', 'valid', 'is') - ('**nsubj**', 'overdrawn', 'account'), ('**cop**', 'overdrawn', 'is'),	R4	**Activity(X,Y,Z):- cop(Z,X),nsubj(Z,Y),not(case(Z,_)).**	Is_card_valid Is_account_overdra wn
	- ('**nsubj**', 'asks', 'customer'), ('**case**', 'cash', 'for'), ('**nmod**', 'asks', 'cash')	R11	**Activity(X,Y,W,Z):- nsubj(X,Y),nmod(X,Z), case(Z,W).**	Asks_customer_for_ cash
	- ('**ccomp**', 'ensure', 'displayed'), ('**nsubjpass**', 'displayed', 'message'),	R22	**Activity(X,Y,W,Z) -: csubjpass(Z,X), dobj(X,Y)**	Ensure_rejection_m essage_displayed
	- ('**csubjpass**', 'dispensed', 'ensure') , ('**neg**', 'dispensed', 'not'), ('**dobj**', 'ensure', 'cash'),	R17	**Activity(X,Y,W,Z) -: csubjpass(Z,X), dobj(X,Y) neg(Z,W)**	Ensure_cash_not_disp ensed
	- ('**nsubjpass**', 'returned', 'card'), ('**ccomp**', 'ensure', 'returned').	R22	**Activity(X,Y,Z):- isverb(X), nsubjpass(Z,Y),ccomp(X,Z), not(dobj(X,Y))**	Ensure_card_returned

In rule "R21", we set the value of its third parameter to "system" at the beginning since the following dependencies are absent: *xcomp* and *nsubj* or *nsubjpass*. In the Prolog rules, *X* is the verb, and *Y* and *Z* are involved in the action *X*. In every rule since *Y* and *Z* are nouns in the most of rules, we look for their compound part by using the compound or amod dependency such as the activities extracted in the table: "*Ensure_rejection_message_displayed*", and "*Fail_login_attempt_system*".

2.2.5. Generation of activity diagram

To constitute the UML activity diagram, we have to define the conditions to insert diagram controls like fork edge, decision node, and join node. These conditions are based on a number of actions extracted from the given acceptance criteria. Table 3 illustrates the rules followed to insert the controls in the generated activity diagram.

Table 3. Symbols detection Rules

Symbols		Rules
Fork		In a case where the number of actions that trigger the activity is just one, and the number of resulting activities is more than one.
Join		In a case where the number of actions that trigger the activity is more than one, and the number of resulting activities is just one.
Decision node		In a case the trigger action contains number and condition indicator such as operator like "*more, less*". When there are two acceptance criteria with opposite actions, then we use the decision node, but only if both acceptance criteria have the same context, i.e., the "*given*" part.

To insert a decision node, we must have a condition and two or more branches depending on the condition. In our case, for example, if we have two acceptance criteria (AC) with the same "*given*" part, and in "*when*" part there is a condition; in the first AC the condition is satisfied and in the second it is not satisfied. In this case, we use the decision node.

The question is “How to determine that the condition is not satisfied in the second AC?”. Depending on the action extracted, we sometimes have a negation that clearly shows the opposition, in other cases, we have to use WordNet to determine if there is an opposition between the conditions.

After extracting the elements of the activity diagram, it is then necessary to gather these components to construct the activity diagram. The generation of the activity diagram as an image is done by PlantUml which is a library allowing to generate UML diagrams from a simple text markup language. First, we used the extracted diagram components to automatically create a text file. This file is parsed by PlantUml to generate a corresponding image of the activity diagram. in our case, PlantUml is handled by the Python language.

An activity diagram is automatically generated from several AC. The AC is divided into three parts: *Given*, *When*, and *Then* parts. Our tool allows us to browse and analyze these parts in each AC to automatically generate a corresponding activity diagram. The procedure is as follows: If the AC are independent, that is, the "*given*" part is different in some or all AC or is not similar to the "*then*" parts in other AC. Then, from the starting point of the diagram, we generate control flows to the activities that are represented by these "*given*" parts. However, if the AC are dependent, for example, the "*then*" part of the first AC is similar to a "*given*" part of another AC, then we trace the control flow from the activity of the "*then*" part in the first AC to the activity extracted from "*when*" part corresponding to the "*given*" part of the other AC.

3. RESULTS AND DISCUSSION

To demonstrate our mechanism of generating the activity diagram from a set of acceptance criteria. We applied the defined rules in several files that contain the AC. The evaluation of the approach was achieved through case studies.

3.1. Case study#1

This case study illustrates two features "withdrawal of cash" and "automated teller machine (ATM) transactions" performed by a customer of a bank.

a) Feature: withdrawal of cash

User story 1: As a customer, I want to withdraw cash from my bank account through ATM. So that I may save time.

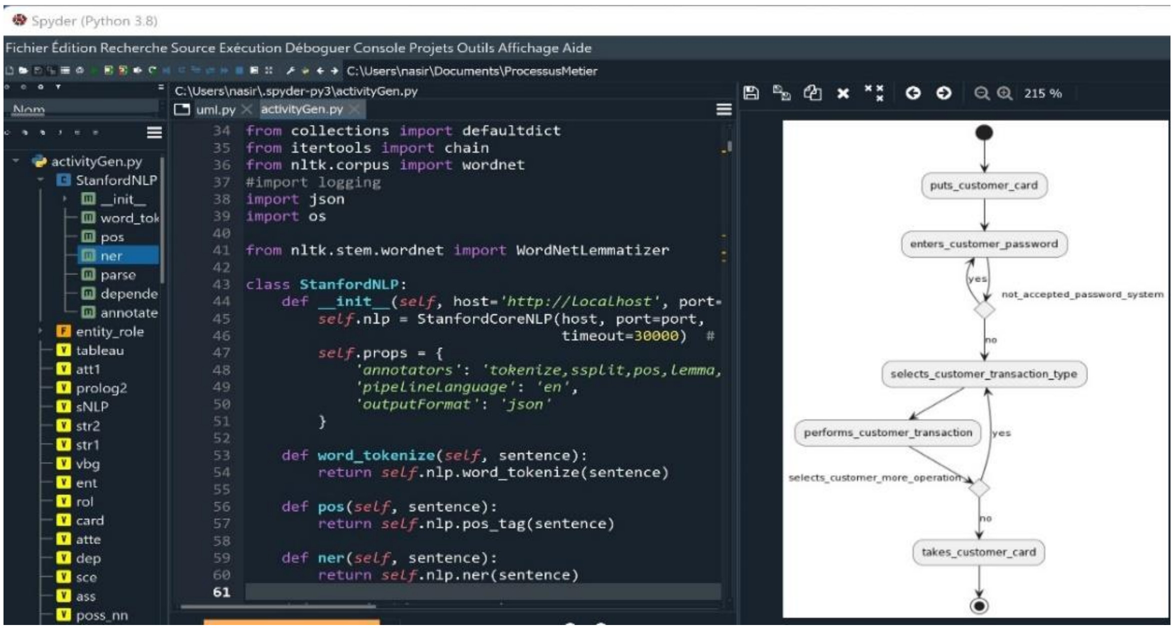
- AC1: *Given* the account is in credit, *when* the customer asks for cash, *then* the account is debited, the cash is dispensed, and the card is returned.
- AC2: *Given* that the account is overdrawn, *when* the customer asks for cash, *then* a rejection message is displayed, cash is not dispensed, and the card is returned.

b) Feature: ATM transactions

User story 2: As a bank customer, I can use an ATM machine to perform transactions.

- AC1: *Given* that the customer puts the card, *when* the customer enters the password, and the password is accepted, *then* the customer selects the transaction type.
- AC2: *Given* the customer puts the card, *when* the customer enters the password, and the password is not accepted, *then* the customer enters the password.
- AC3: *Given* that the customer selects a transaction type, *when* the customer performs the transaction, and the customer selects more operations, *then* the customer selects a transaction type.
- AC4: *Given* that the customer selects the transaction type, *when* the customer performs the transaction, and the customer selects no operation, *then* the customer takes the card.

Figures 6 and 7 respectively show the activity diagram generated from the given acceptance criteria describing the cash withdrawal and ATM transaction processes. Figure 6 illustrates the use of the conjunction "*and*" between activities in the "*then*" part which led to the insertion of the fork node. Because the "*given*" part is not similar to the two acceptance criteria, our tool automatically draws 2 flow controls to "*is_account_in_credit*" and "*is_account_overdrawn*". Our method was unable to detect that these two elements are opposite in order to insert the decision node. Despite the absence of this symbol, the activity corresponding is not missing in the diagram.


Spyder (Python 3.8)
Fichier Édition Recherche Source Exécution Déboguer Console Projets Outils Affichage Aide
C:\Users\nasir\Documents\ProcessusMetier
C:\Users\nasir\.spyder-py3\activityGen.py
uml.py
activityGen.py
Nom
activityGen.py
StanfordNLP
__init__
word_tok
pos
ner
parse
depende
annotate
entity_role
tableau
att1
prolog2
sNLP
str2
str1
vbg
ent
rol
card
atte
dep
sce
ass
poss_nn
215 %
puts_customer_card
enters_customer_password
yes
not_accepted_password_system
no
selects_customer_transaction_type
performs_customer_transaction
yes
selects_customer_more_operation
no
takes_customer_card

```
from collections import defaultdict
from itertools import chain
from nltk.corpus import wordnet
#import logging
import json
import os

from nltk.stem.wordnet import WordNetLemmatizer

class StanfordNLP:
    def __init__(self, host='http://localhost', port=
        self.nlp = StanfordCoreNLP(host, port=port,
                                   timeout=30000)  #
        self.props = {
            'annotators': 'tokenize,ssplit,pos,lemma,
            'pipelineLanguage': 'en',
            'outputFormat': 'json'
        }

    def word_tokenize(self, sentence):
        return self.nlp.word_tokenize(sentence)

    def pos(self, sentence):
        return self.nlp.pos_tag(sentence)

    def ner(self, sentence):
        return self.nlp.ner(sentence)

```

Figure 6. Activity diagram generated from given acceptance ("withdrawal of cash" process)

In Figure 7, because AC1 and AC2 have the same given part, and the "*then*" part of AC1 is similar to the given part of AC3 and AC4, then we have a single arrow from the entry point of the diagram, therefore "*select_customer_transaction_type*" is not an entry of the diagram. Two new decisions were generated due to the extraction of the opposite elements, first "*accepted_password_system*" and "*not_accepted_password_system*", then "*select_customer_more_operation*" and "*select_customer_no_operation*".

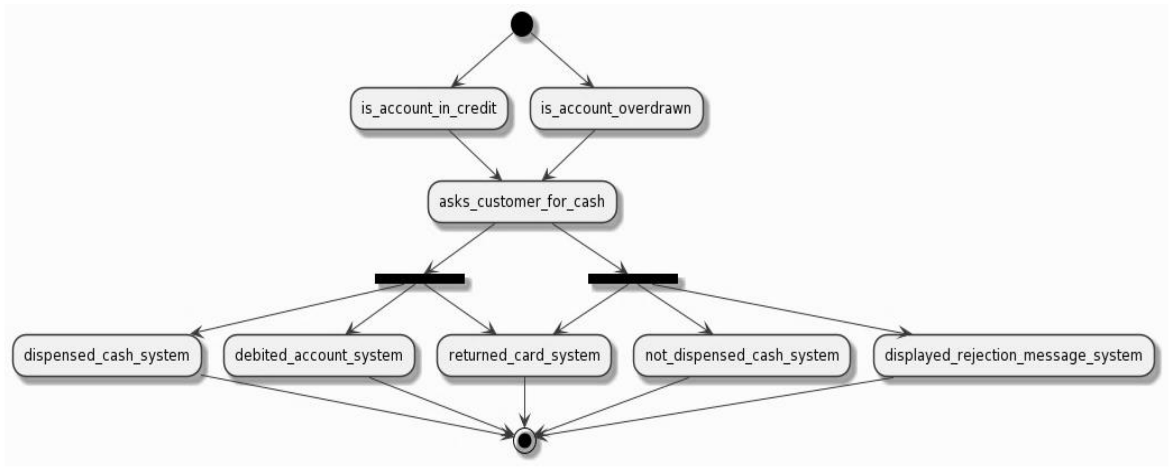


Figure 7. Activity diagram generated from given acceptance criteria ("ATM transactions" process)

3.2. Case study#2

This case study illustrates two features "Log in" and "Log out" to access a page of an application.

a) Feature: log in

User story 1: As a registered user, I want to log in with the correct username and password so that the system can authenticate me, and I can trust it.

- AC1: *Given* that the registered user is logged out, *when* he navigates to the login page, and enters his incorrect username and password, *then* log in fails with an error message that specifies that the username or password was wrong.
- AC2: *Given* that the registered user is logged out, *when* he navigates to the login page, and enters a correct username and password, *then* his session is loaded in less than eight seconds.

User story 2: As a user, I want to receive a message error when I leave login or password fields empty.

- AC3: *Given* that the registered user is logged out, *when* he navigates to the login page, and he left username and password fields empty, and clicks on log in, *then* error message should be displayed, and login attempt should fail.

b) Feature: log out

User story 1: As a user, I want a log out feature in my account, so that I can end the session.

- AC1: *Given* that the user is on the application page, *when* the user clicks on the logout link, *then* the user must be logged out from all active sessions of the application.
- AC2: *Given* that the user is on the application page, *when* more than 30 minutes of inactivity has passed and the user reloads the page, *then* the user must be logged out from all active sessions of the application and redirected to the login page.

User story 2: As a user, I want to be logged out from the application, so that the logout button is hidden when I am not logged in.

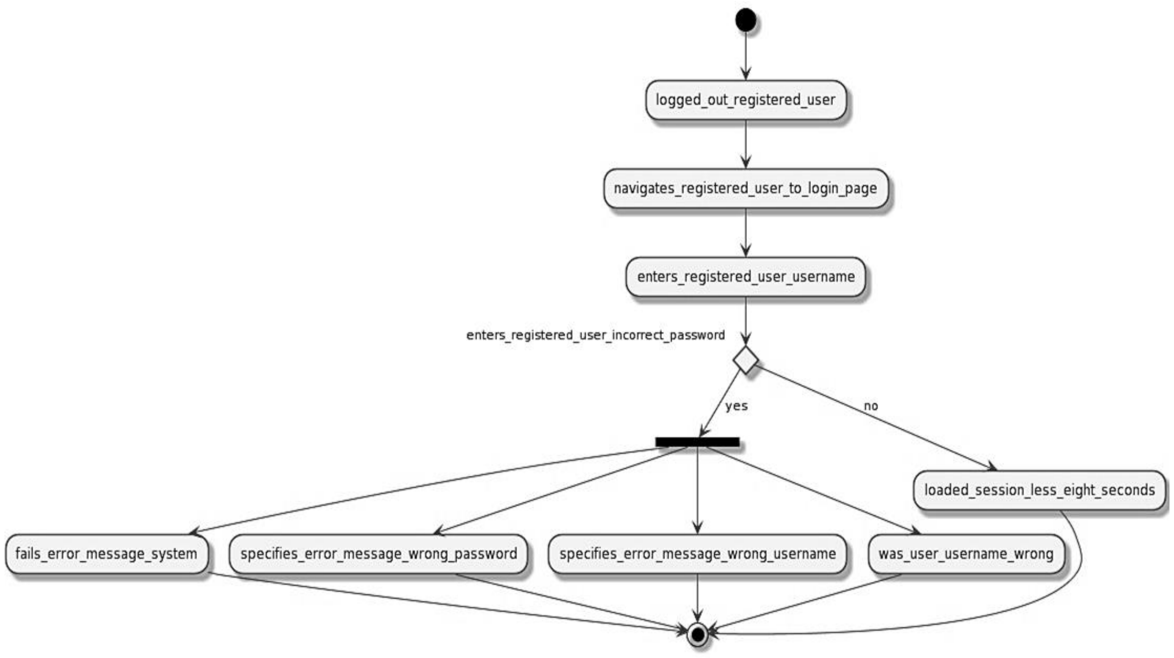
- AC3: *Given* the user is logged out, *when* the user is on the application page, *then* the user will not see the logout button.

User story 3: As a user, I want to receive notifications when there is unsaved work in my account before I log out, so that I can save my work and not lose it.

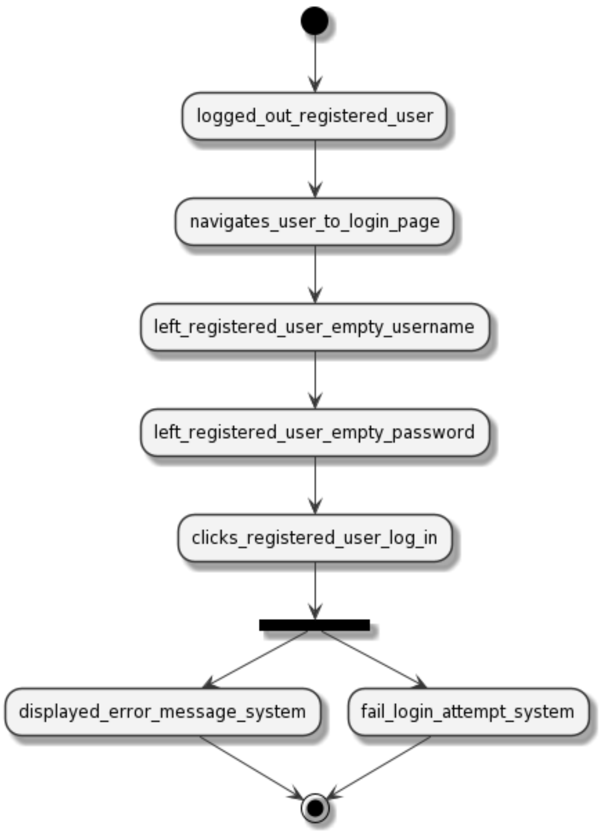
- AC4: *Given* a user selects log out, *when* there is unsaved work, *then* the user should not be logged out and warn the user that their unfinished work will be lost.
- AC5: *Given* a user selects log out, *when* there is no unsaved work, *then* the user should be successfully logged out.

Figure 8 depicts the generated activity diagram corresponding to the previous acceptance criteria. Figure 8(a) shows the authentication process of AC1 and AC2, and Figure 8(b) depicts the activities of AC3. Our approach takes into account the order of actions in the "*when*" and the "*given*" parts, as well as the detection of conditions as in Figure 8(a), the Wordnet Library is used to determine the antonyms such as between "incorrect" and "correct" words in AC1 and AC2.

Figure 9(a) illustrates the use of the condition indicator as "more" in our approach, and also the conjunction "and" between activities in the "then" part which led to the insertion of the fork node. Figure 9(b) shows that our approach handles the preposition "on" and the compound noun "application page", as well as the negation "not see". As in Figure 8(a), in Figure 10 our approach inserts the decision node because of the antonyms detection of "unsaved" and "no unsaved" in AC1 and AC2.

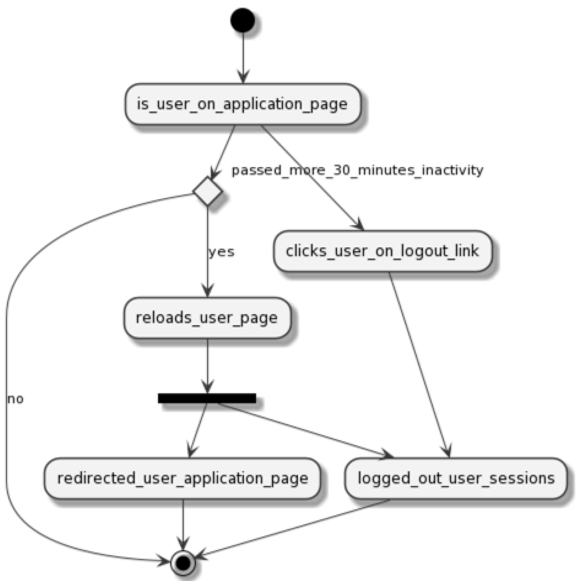

logged_out_registered_user
navigates_registered_user_to_login_page
enters_registered_user_username
enters_registered_user_incorrect_password
yes
no
loaded_session_less_eight_seconds
fails_error_message_system
specifies_error_message_wrong_password
specifies_error_message_wrong_username
was_user_username_wrong

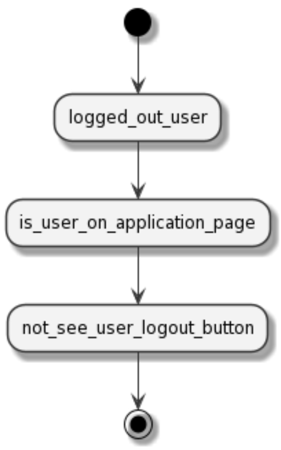
(a)


logged_out_registered_user
navigates_user_to_login_page
left_registered_user_empty_username
left_registered_user_empty_password
clicks_registered_user_log_in
displayed_error_message_system
fail_login_attempt_system

(b)

Figure 8. Activity diagram generated from (a) AC 1 and AC2, and (b) AC3 ("log in" process)


is_user_on_application_page
passed_more_30_minutes_inactivity
yes
no
clicks_user_on_logout_link
reloads_user_page
redirected_user_application_page
logged_out_user_sessions



(a) (b)

Figure 9. Activity diagram generated from (a) AC 1 and AC2, and (b) AC3 ("log out" process)

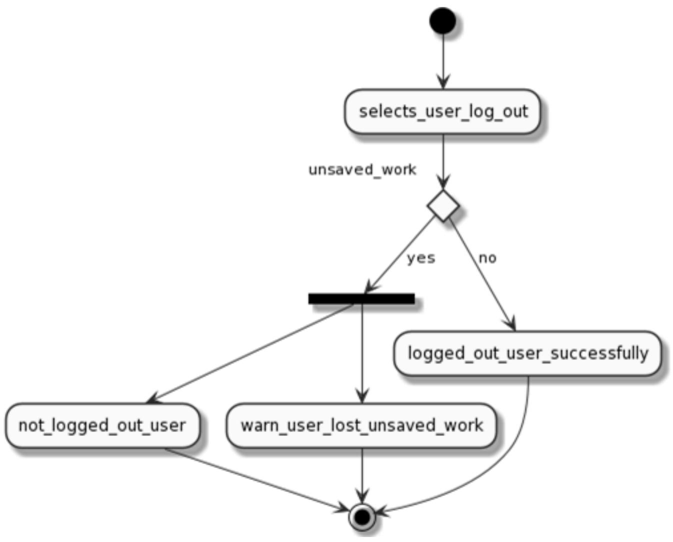


Figure 10. Activity diagram generated from AC 4 and AC5 ("log out" process)

3.3. Performance evaluation

To evaluate the performance of our approach, we used some case studies with different acceptance criteria, and to increase the number of tests, we modified them by changing the acceptance parts to test different sentence structures. To assess the results obtained. An activity diagram was constructed manually for each file containing the user story with several acceptance criteria, and then both models, the manual and the automatic were compared.

We noted that our approach extracts the activities requested either the context, the triggering action, or the activities results. The generation of the activity diagram based on the acceptance criteria revealed its efficiency. Tables 4 and 5 show a comparison of our approach to extracting activity diagram components versus the manual approach with accuracy.

Table 4 illustrates that the proposed technique was capable of retrieving almost all artifacts efficiently. The total number of activities is 50 activities including 33 for the first case study and 17 for the second case study. Our approach extracts 2 activities that are included in the detected activities, so we thought of refining the results and detecting the inclusion of activities, and we developed a python code. Among these 2 activities, we find "*lost_unsaved_work*" in Figure 10 and "*wrong_username*" in Figure 8. As our approach allows us to refine the results in the case of redundancy caused by the inclusion of activities, this can only assure us about the effectiveness of our method. Concerning the first FN, in Table 5, it is the missing "incorrect" adjective for the username in the "*enters_registred_user_username*" action in Figure 7. In the second case study, when extracting the symbols, it appeared that the decision symbol is missing, this symbol is the only component missing in this case study.

Table 4. Total number of activities results versus the manual approach

Metrics	Total number of activities
Our approach	50
Manually	48

Table 5. Evaluation results of our approach

Metrics	TP	FN	FP	Recall (%)	Precision (%)	Accuracy (%)
Our approach	47	1	2	97%	94%	94%

Through the rules of symbol extraction, and the construction of the activity diagram, we were able to group and graphically represent the extracted activities in the activity diagram automatically and with a minimum of errors or missing components. Finally, it must be pointed out that our approach has a few limitations, especially in the detection of antonyms using a negation concept or the wordnet library in order to generate more decision nodes, so that the diagram can be more easily comprehensible and not too cluttered by the sequence of activities. Nevertheless, it should be noted that our approach focuses on the management of conjunctions and compound nouns as well as negation, which are important points for obtaining the desired results.

3.4. Discussion

In this part, we present a discussion based on an analytical survey. The state of art approaches [13], [14] that are based on the same tool inputs as our approach, namely acceptance criteria, and user stories, focus mainly on test case generation. In [18] the BPMN generation was performed from textual requirements, as for the user stories, the approach just generates the BPMN operations. Moreover, they do not use acceptance criteria with user stories. Their paper is oriented to generate entities and attributes that constitute the relationship model between entities. Furthermore, in our previous works [27], [28], we generated a class diagram, use case diagram, and package diagram, from the user stories using NLP techniques, and created an ontology to refine the generated models. But in this work, we focused on the business process. We accompanied the user stories with acceptance criteria to give more details about the business process since the description of the business process in the use case diagrams is insufficient and incomplete. In other approaches, the transformation from model to model is performed. For this, most authors have defined a set of transformation rules using the ATL language.

4. CONCLUSION AND FUTURE WORKS

Requirements analysts manually analyze requirements in natural language to extract the elements that makeup UML diagrams. Manual analysis requires a lot of time and effort, so automated support is needed. In this paper, we proposed an approach to ease the process of natural language requirements analysis and to obtain UML diagrams from natural language textual requirements.

As agile requirements, we used user stories with acceptance criteria that follow the model: "*Given, When, Then*". In this model, we employed several ANDs for several requirements. These requirements are processed using NLP techniques in order to generate an activity diagram. In our tests, we were not only satisfied with a single acceptance criterion related to a user story, but we used several scenarios. To perform the diagram generation, we started by defining a list of rules for extracting activities. These rules are developed using the Prolog language. Regarding the control flows between activities in the activity diagram, we analyzed the acceptance criteria, we focused on the "*When*" part first to detect the conditions that will be needed to generate the decision node, and then to order the actions according to their mention in the "*When*" part. In the future, we plan to use deep learning models, first to improve the detection of opposite actions or conditions in the "*When*" part of AC that is quite complex to detect with the wordnet, and therefore generate the decision node. Second, to detect synonyms between activities or actions. In addition, we will focus on the automatic generation of code related to business services.

REFERENCES

[1] M. A. Akbar, A. Alsanad, S. Mahmood, A. A. Alsanad, and A. Gumaei, "A systematic study to improve the requirements engineering process in the domain of global software development," *IEEE Access*, vol. 8, pp. 53374–53393, 2020, doi: 10.1109/ACCESS.2020.2979468.

[2] H. A. Reijers, "Business process management: The evolution of a discipline," *Computers in Industry*, vol. 126, Apr. 2021, doi: 10.1016/j.compind.2021.103404.

[3] S. O. Barraood, H. Mohd, and F. Baharom, "A comparison study of software testing activities in Agile methods," in *Knowledge Management International Conference (KMICe) 2021*, 2021, pp. 130–137.

[4] G. Lucassen, M. Robeer, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper, "Extracting conceptual models from user stories with Visual Narrator," *Requirements Engineering*, vol. 22, no. 3, pp. 339–358, 2017, doi: 10.1007/s00766-017-0270-1.

[5] E. A. Abdelnabi, A. M. Maatuk, T. M. Abdelaziz, and S. M. Elakeili, "Generating UML class diagram using NLP techniques and heuristic rules," in *2020 20th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, Dec. 2020, pp. 277–282, doi: 10.1109/STA50679.2020.9329301.

[6] V. B. R. Vidya Sagar and S. Abirami, "Conceptual modeling of natural language functional requirements," *Journal of Systems and Software*, vol. 88, pp. 25–41, Feb. 2014, doi: 10.1016/j.jss.2013.08.036.

[7] H. Herchi and W. Ben Abdessalem, "From user requirements to UML class diagram," *arXiv preprint arXiv:1211.0713*, Nov. 2012.

[8] N. Bashir, M. Bilal, M. Liaqat, M. Marjani, N. Malik, and M. Ali, "Modeling class diagram using NLP in object-oriented designing," in *2021 National Computing Colleges Conference (NCCC)*, 2021, pp. 1–6, doi: 10.1109/NCCC49330.2021.9428817.

[9] J. S. Thakur and A. Gupta, "Automatic generation of analysis class diagrams from use case specifications," *arXiv preprint arXiv:1708.01796*, Aug. 2017.

[10] M. Elallaoui, K. Nafil, and R. Touahni, "Automatic transformation of user stories into UML use case diagrams using NLP techniques," *Procedia Computer Science*, vol. 130, pp. 42–49, 2018, doi: 10.1016/j.procs.2018.04.010.

[11] M. Javed and Y. Lin, "Iterative process for generating ER diagram from unrestricted requirements," in *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018, pp. 192–204, doi: 10.5220/0006778701920204.

[12] A. M. Maatuk and E. A. Abdelnabi, "Generating UML use case and activity diagrams using NLP techniques and heuristics rules," in *International Conference on Data Science, E-learning and Information Systems 2021*, Apr. 2021, pp. 271–277, doi: 10.1145/3460620.3460768.

[13] J. Fischbach, A. Vogelsang, D. Spies, A. Wehrle, M. Junker, and D. Freudenstein, "SPECMATE: Automated creation of test cases from acceptance criteria," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, Oct. 2020, pp. 321–331, doi: 10.1109/ICST46399.2020.00040.

[14] P. Pandit and S. Tahiliani, "AgileUAT: A framework for user acceptance testing based on user stories and acceptance criteria," *International Journal of Computer Applications*, vol. 120, no. 10, pp. 16–21, Jun. 2015, doi: 10.5120/21262-3533.

[15] R. Sibal, P. Kaur, and C. Sharma, "Prioritization of user story acceptance tests in agile software development using meta-heuristic techniques and comparative analysis," in *Towards Extensible and Adaptable Methods in Computing*, Singapore: Springer Singapore, 2018, pp. 43–55.

[16] B. Maqbool *et al.*, "A comprehensive investigation of BPMN models generation from textual requirements—techniques, tools and trends," in *ICISA 2018: Information Science and Applications 2018*, 2019, pp. 543–557.

[17] S. Gulia and T. Choudhury, "An efficient automated design to generate UML diagram from natural language specifications," in *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, Jan. 2016, pp. 641–648, doi: 10.1109/CONFLUENCE.2016.7508197.

[18] M. Javed and Y. Lin, "iMER: Iterative process of entity relationship and business process model extraction from the requirements," *Information and Software Technology*, vol. 135, Jul. 2021, doi: 10.1016/j.infsof.2021.106558.

[19] Y. Rhazali, Y. Hadi, and A. Mouloudi, "A new methodology CIM to PIM transformation resulting from an analytical survey," in *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development*, 2016, pp. 266–273, doi: 10.5220/0005690102660273.

[20] K. Kluza and G. J. Nalepa, "A method for generation and design of business processes with business rules," *Information and Software Technology*, vol. 91, pp. 123–141, Nov. 2017, doi: 10.1016/j.infsof.2017.07.001.

[21] T. Paszun, P. Wiśniewski, K. Kluza, and A. Ligęza, "Automated generation of business process models using constraint logic programming in python," in *Proceedings of the 2019 Federated Conference on Computer Science and Information Systems, FedCSIS 2019*, Sep. 2019, pp. 733–742, doi: 10.15439/2019F174.

[22] A. Bouzidi, N. Haddar, M. Ben Abdallah, and K. Haddar, "Deriving use case models from BPMN models," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, Oct. 2017, pp. 238–243, doi: 10.1109/AICCSA.2017.49.

[23] B. Aysolmaz, H. Leopold, H. A. Reijers, and O. Demirörs, "A semi-automated approach for generating natural language requirements documents based on business process models," *Information and Software Technology*, vol. 93, pp. 14–29, Jan. 2018, doi: 10.1016/j.infsof.2017.08.009.

[24] L. Erazo, E. Martins, and J. G. Greghi, "MARITACA: From textual use case descriptions to behavior models," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun. 2017, pp. 83–90, doi: 10.1109/DSN-W.2017.33.

[25] F. Gilson and C. Irwin, "From user stories to use case scenarios towards a generative approach," in *2018 25th Australasian Software Engineering Conference (ASWEC)*, Nov. 2018, pp. 61–65, doi: 10.1109/ASWEC.2018.00016.

[26] A. Ivanchikj, S. Serbout, and C. Pautasso, "From text to visual BPMN process models," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Oct. 2020, pp. 229–239, doi: 10.1145/3365438.3410990.

[27] S. Nasiri, Y. Rhazali, M. Lahmer, and N. Chenfour, "Towards a generation of class diagram from user stories in Agile methods," *Procedia Computer Science*, vol. 170, pp. 831–837, 2020, doi: 10.1016/j.procs.2020.03.148.

[28] S. Nasiri, Y. Rhazali, M. Lahmer, and A. Adadi, "From user stories to UML diagrams driven by ontological and production model," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 6, 2021, doi: 10.14569/IJACSA.2021.0120637.

BIOGRAPHIES OF AUTHORS

Samia Nasiri is a Ph.D. student at LMMI Laboratory of ENSAM, Moulay Ismail University of Meknes, Morocco. She received an engineering degree in computer science from the National School of Applied Sciences of Oujda, Morocco. She is a professor in the higher school of technology of Meknes. Her research interests concern software engineering, natural language processing, and artificial intelligence. She can be contacted at email: nasiri.samia@gmail.com.

Amina Adadi is an associate professor at the High School of Technology, Moulay Ismail University, Meknes, Morocco. She received an engineering degree in computer science from the National School of Applied Sciences of Fez in 2012, and a Ph.D. degree in computer science from Sidi Mohammed Ben Abdellah University, Fez, Morocco, in 2017. Her current research interests include artificial intelligence, machine learning, software architecture, and semantic web services. She can be contacted at email: Amina.adadi@gmail.com.

Mohammed Lahmer is a professor-researcher at the High School of Technology, Moulay Ismail University. He received his Ph.D. degree in computer science from ENSIAS Mohammed V University in 2008. He received his engineering degree in computer science from ENSIAS in 1996. He has over twenty years of teaching experience and several scientific publications in top IEEE conferences and journals. His research interests lie in the field of security, and software engineering. He can be contacted at mohammed.lahmer@gmail.com.