Towards the Improvement of Natural Language Requirements Descriptions: The C&L Tool*

Edgar Sarmiento-Calisaya
Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
esarmientoca@unsa.edu.pe

Edward Hinojosa Cárdenas
Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
ehinojosa@unsa.edu.pe

Victor Cornejo-Aparicio
Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú
vcornejo@unsa.edu.pe

Guina Sotomayor Alzamora
Universidad Nacional del Altiplano de Puno
Puno, Perú
gsotomayor@unap.edu.pe

ABSTRACT

A natural language-based requirements specification tends to be full of requirements that are ambiguous, unnecessarily complicated, missing, wrong, duplicated or conflicting. Poor quality requirements often compromise the subsequent software construction activities (e.g. planning, design, coding or testing). A strategy for requirements quality evaluation should enable a faster requirements analysis, highlight defect indicators and incorporate also fix recommendations to help practitioners to effectively improve their requirements. In this paper we brief describe a Natural Language Processing and Petri-Net strategy for automated analysis of scenario-driven requirements named C&L prototype tool. The C&L evaluates structural (Static analysis) aspects of scenarios and behavioral aspects (Dynamic analysis) of equivalent Petri-Nets. The feasibility of the C&L is evaluated in four projects described as use cases, which indicates promising results (the overall precision was 93.5% and the recalls were perfect).

CCS CONCEPTS

• **Software and its engineering → Requirements analysis**;

KEYWORDS

Requirement, scenario, use case, analysis, quality, automation

ACM Reference Format:
Edgar Sarmiento-Calisaya, Edward Hinojosa Cárdenas, Victor Cornejo-Aparicio, and Guina Sotomayor Alzamora. 2020. Towards the Improvement of Natural Language Requirements Descriptions: The C&L Tool. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3341105.3374028

*Produces the permission block, and copyright information

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3374028

1 INTRODUCTION

Requirements Engineering (RE) activities are closely related to the good quality of software [6] [23] [29] [39], i.e., the definition of requirements and their analysis are key factors for successful software development companies. However, current practices often lead to poor quality of requirements. This problem arises because requirements are written using natural language and informally *analyzed* [18].

Requirements analysis improves the quality of a Software Requirements Specification (SRS) document, and it is subdivided into verification and validation. These activities are generally carried out through a manual procedure for reading documents, *finding defects* and *addressing* these problems. Therefore, analysis is still an expensive process, which requires great effort and takes a lot of time. Thus, the automation of these activities is a challenging topic.

Since scenario-based representations are often stated in natural language, they are widely used in requirements engineering because it helps developers, engineers and other stakeholders to better understand the requirements and its interface with the environment [23]. In this context, SRSs are represented as a collection of scenarios and described by specific flows of events. The most prominent languages to write scenarios are use case descriptions [9] [11] [38] [44], scenarios [23] or their variations.

However, scenarios can then hardly be automatically analyzed due to natural language lack formal semantics to support the automated analysis of structural and behavioral properties.

The early identification of defects and the removal of their causes can greatly improve the quality of scenarios; contributing to increase software reliability, to improve development productivity, to reduce the inconsistency risks between requirements and other artifacts generated from them, and to keep the software development within budget and delivery time [6].

Many researchers have begun to see potential benefits from adding Natural Language Processing (NLP) [3] [4] [10] [16] [20] and Petri-Nets [21] [32] [38] capabilities to requirements analysis approaches. The challenge is to create NLP and Petri-Net modules that help automatically detect *defects* occurring in individual scenarios or a set of scenarios and their relationships.

In this paper we describe an attempt towards automation of requirements analysis. To this purpose the C&L (Scenarios & Lexicons) prototype tool: (1) Parses textual scenarios into structured

scenarios; (2) Annotates structured scenarios with relevant information provided by Natural language Processing (NLP) tools; (3) Discover sequential and non-sequential relationships among scenarios; (4) Derives Petri-Nets from structured scenarios; (5) Discovers certain types of defects in scenarios; and (6) Advises fix recommendations to repair defects. The C&L supports the most existing scenario and use case templates and enables a faster requirements analysis or serves as a complement to RE activities.

This paper is organized as follows. Section 2 describes the scenario language and its *verification heuristics* implemented in the C&L, and it describes the implementation and functionalities of the C&L. Section 3 presents the approach implemented in the C&L. Section 4 describes the evaluation. Section 5 describes some related work to our proposal, Finally, Section 6 presents the conclusions.

2 BACKGROUND

In this section we will describe the language of scenarios used in modeling requirements, also, we describe the verification and validation heuristics to evaluate the quality of scenarios. The C&L tool, which implements the method proposed in this work, is also presented in this section.

2.1 Scenario

Cockburn [11] and Glinz [19] define scenarios as "an ordered set of interactions between a system and a set of actors external to the system – *a use case description*". In contrast, Leite et al. [23] define scenario as "a description technique that is both process-focused and user centric", i.e., it describes *"situations in the macrosystem and not only a sequence of interactions between users and a system"*. Besides, it presents characteristics to represent relationships among scenarios.

There are a wide variety of scenario representations in the literature, each one with quite different purposes. Therefore, scenarios can take many forms and provide various types of information on different levels of abstraction and formalism.

Based on the scenario language proposed by Leite et al. [23] and studies about use case templates reported in [36] [40] we define a template for writing scenarios that emphasizes on specifying the main flow and the alternative flows sections. This template makes use of a single or two-column format to specify the textual descriptions. Fig. 1 details the template (and its sections) that supports the scenario language [23] and the use case templates based on Cockburn [11]. Frequently, statements within scenario sections are described by simple sentences centered on the main verb (Fig. 2), i.e., *subject + action-verb + object* such as demonstrated in [1] [9] [11] [13] [23] [36] [38] [44].

In order to facilitate the transformation into domain models and other software artifacts (e.g. test cases), several templates (reported in [36] and [40]) are in favor of:

(1) Writing the main flow events in a simple, conditional, loop or parallel way;
(2) The use of a numbered style format in the main and alternative flows;
(3) The use of some specific keywords such as *IF-THEN*, *DO-UNTIL*, *ABORT*, *RESUME* and *GO TO*;

TITLE/NAME: <An identification of the scenario>
GOAL/DESCRIPTION: <Brief text that express the purpose of the scenario>
CONTEXT:
PRE-CONDITION: <What should be true before the scenario starts>
POST-CONDITION: <What should be true after the basic flow ends>
ACTOR: <Person, another system, device, system component or organization structure directly involved with the situation>
[RESOURCE: <Physical element or data used to achieve its goal>]
EPISODES/ MAIN FLOW: <The most common and successful flow of events>
A sentence is described using simple, conditional, loop or parallel (or concurrent or indistinct order) structure and using special keywords.
Sentences are numbered sequentially:
<Step> <Action or name of sub-scenario>
1 <Sentence>
ALTERNATIVES / EXCEPTIONS: <Describes a deviation in another flow of events>
- Alternate: less-frequented flow to achieve the goal.
- Exception: flow that leads to an obstacle to achieve the goal.
An alternative flow always depends on a condition occurring in a specific step of the main flow and is composed of a sequence of numbered sentences:
<Step><Ref> <Condition causing branching>:
[<Step><Ref><Step>] <Action or name of sub-scenario>
1.a <Cause>:
1.a.1 <Solution>
2E <Cause>:
2E1 <Solution>

Figure 1: Template for writing scenarios.

(4) That alternative (alternate/exception) numberings should match the numbers (steps) in the main flow;
(5) That every alternative flow must return to some specific step of the main flow or finish the scenario [40] [42].

Other approaches [9] [23] [34] use *Resource/Variable/Business Item* that holds the data during the execution of a scenario/use-case.

[<SUBJECT>] <VERB> <OBJECT>
Customer examines the bid
[<SUBJECT>] <VERB> <OBJECT> <PREPOSITIONAL PHRASE>
The broker system sends the bid to the customer
[<SUBJECT>] <VERB> <PREPOSITIONAL PHRASE>
The Broker System asks for the Customer's login information

Figure 2: Common guidelines for writing scenario statements.

We therefore state a definition from [11] [19] [23] [36] [40]:

Definition 1 (Scenario): A *scenario* is a partial description of the application behaviour that occurs in a specific *context*, with all necessary *resources* and *actors*, and must satisfy a *goal* that is achieved by performing *events* (each guarded by conditions or restricted by constraints) occurring in its *episodes* and *alternative* flows.

An *event* is an episode sentence or an alternative solution step, it describes an action or interaction involving users, organizations, environment, system components, resources and the system. A *condition* is an actor, resource or system state.

Fig. 3 depicts an abstract conceptual model for scenarios. The *episodes* describe the main flow of actions. *Alternative* flows can

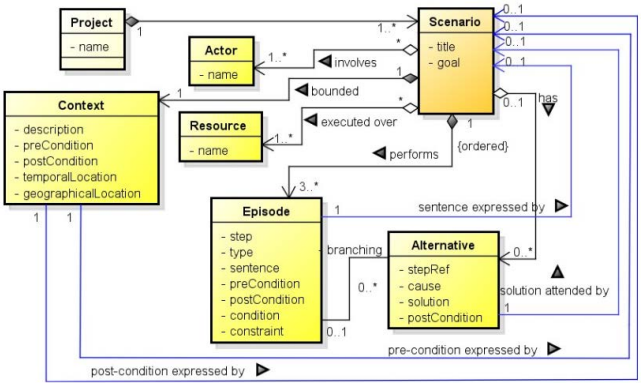

Figure 3: Scenario conceptual model.

arise during the execution of episodes, which indicate that there exists either a less-frequented way to achieve the goal (*alternate*) or an obstacle to achieve the goal (*exception*).

In order to parsing textual scenarios into structured scenarios we define a grammar based on partial Extended-BNF (Fig. 4). It supports the existing templates for describing scenarios or use cases.

Fig. 5 presents the use case Check Balance in the ATM System [12]. The goal is: User wants to check their account balance before withdrawing money.

2.2 Scenario Analysis

Evaluating the quality of an SRS involves aspects related to *verification* and *validation* of software artifacts produced in the requirements engineering process, i.e., scenario descriptions. The quality of scenarios can be evaluated by *verification heuristics* that search indicators of violation (*Defects*) of *unambiguity, completeness* and *consistency* properties.

Proposals for a quality model for scenarios and its verification heuristics are detailed in [30] [31]. A set of heuristics were implemented in the C&L tool.

When the result after performing a *verification heuristic* is opposite to an expected result, a *defect* (indicator of violation) must be reported to the requirements engineer. *Defects* are categorized as: Information, Warning or Error. *Information* reveals that the requirements engineer may have forgotten to specify some information related to a scenario element. *Warning* reveals that the requirements engineer may have introduced some confusing information or forgotten to inform and important scenario element. *Error* reveals that the requirements engineer may have introduced wrong information related to a scenario element. The presence of defects is a strong indication, although not conclusive, of incorrectness that must be fixed.

2.2.1 Verify Unambiguity. A specification is *unambiguous* if and only if, every requirement stated therein has only one interpretation [37]. The verification includes:

(1) Check that a sentence does not contain vague terms (e.g. adaptability, additionally);
(2) Check that a sentence does not contain subjective words, i.e., comparative/superlative adverbs/adjectives (e.g. similar, better);
(3) Check that a sentence does not contain optional words (e.g. as desired, at last);
(4) Check that a sentence does not contain weak terms (e.g. can, preferred);
(5) Check that a sentence has not multiple action verbs or subjects, i.e., conjunction or disjunction (e.g. and, or);
(6) Check that a sentence does not contain implicit words (e.g. anyone, he, her);
(7) Check that a sentence uses quantification words in a clear way (e.g. all, any, few);

2.2.2 Verify Completeness. A specification is *complete* if all relevant requirements are present and each requirement is fully developed [5]. The verification includes:

(1) Check that Title defines exactly one situation (a verb in infinitive (base) form + an object) [11] [23];
(2) Check that Episode-Sentence is described from user point of view (Subject + present simple tense and active form of verb + Object), or by another scenario (infinitive verb – base form + Object) [10] [11] [23] [44];
(3) Check that Alternative-Solution Step is described: from user point of view (present simple tense and active form of verb + Object), or by another scenario (infinitive – base form + Object). Optionally, it contains a Subject [10] [11] [23] [44].
(4) Check that nested IF statement is not used in a Conditional Episode, i.e., it can confuse the user and be difficult to read [10] [12];
(5) Check that alternative is handled by a simple action [23], i.e., if the interruption is treated by a sequence of steps (>2), this sequence should be extracted to a separate scenario [10];
(6) Check that every alternate flow returns to a specific episode of the main flow and an exception finishes the scenario [42].
(7) Check the completeness of every scenario component/section;
(8) Check the syntax of each component as established in the scenario model;
(9) Check that every Actor participates in at least one episode [23];
(10) Check that every Resource is used in at least one episode [23];
(11) Check that every Subject mentioned in episodes is an Actor, Resource [23] or the System [44];
(12) Check that every Subject mentioned in alternatives is an Actor, Resource [23] or the System [44];
(13) Ensure that step numbering between the main flow and alternative flow are consistent [24];
(14) Check the existence of more than two and less to 10 episodes per scenario [10] [11] [23];
(15) Check that the Title describes the Goal;
(16) Check that every included scenario (Pre-condition, Post-condition, Episode sentence, Alternative solution, Constraint) exists within the set of scenarios [23];
(17) Ensure that actions present in the Pre-conditions are already performed [23];
(18) Check that Episode coincidence only takes place in different situations [23];

<scenario> ::= TITLE: <Title> + GOAL: <Goal> + CONTEXT: <Context> + [RESOURCE: {<Resource>}1^N]+ ACTOR: {<Actor>}1^N + EPISODES: <Episodes> +
ALTERNATIVES: {<Alternative>}
<Context> ::= [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>] + [GEOGRAPHICAL LOCATION: <Geographical Location>] +
[TEMPORAL LOCATION: <Temporal Location>]
<Geographical Location> ::= <Item> | <Geographical Location> <connective> <Geographical Location>
<Temporal Location> ::= <Item> | <Temporal Location> <connective> <Temporal Location>
<Pre-condition> ::= <State> | <Title> | <Pre-condition> <connective> <Pre-condition>
<Post-condition> ::= <State> | <Title> | <Post-condition> <connective> <Post-condition>
<Resource> ::= <Item> + {<Constraint>}
<Actor> ::= <Item>
<Episodes> ::= <Group> | <Episodes> <Group>
<Group> ::= <Sequential Group> | <Non-Sequential Group>
<Sequential Group> ::= <Episode><Episode> | <Sequential Group><Episode>
<Non-Sequential Group> ::= {<Episode>} # <Episode Series> # {<Episode>}
<Episode Series> ::= <Episode> <Episode> | < Episode Series><Episode>
<Episode> ::= (<Simple Episode> | <Conditional Episode> | <Optional Episode> | <Loop Episode>) + [PRE-CONDITION: <Pre-condition>] +
[POST-CONDITION: <Post-condition>] + {<Constraint>}
<Simple Episode> ::= <Step> + (<delimiter> | white-space) + <Episode Sentence>
<Conditional Episode> ::= <Step> + (<delimiter> | white-space) + IF <Condition> THEN <Episode Sentence>
<Optional Episode> ::= <Step> + (<delimiter> | white-space) + "[" + <Episode Sentence>
<Loop Episode> ::= <Step> + (<delimiter> | white-space) + ((DO | REPEAT) <Episode Sentence> (WHILE | UNTIL) <Condition> | (WHILE | UNTIL) <Condition>
(DO | REPEAT) <Episode Sentence> | FOR-EACH <Item> (DO | REPEAT) <Episode Sentence>)
<Alternative> ::= <Step> + <Ref> + (<delimiter> | white-space) + (IF + <Cause> + THEN | <Cause>) + [:] +
{[(<Step> | <Step> + <Ref> + <Ref>) + (<delimiter> | white-space)] + <Solution-Step>}1^N + [POST-CONDITION: <Post-condition>]
<Step> ::= {Digit}$_1^N$
<Ref> ::= (<delimiter> + {Digit}$_1^N$ | [<delimiter>] + Letter)
<delimiter> ::= (. | :)
<Condition> ::= <Atomic Sentence> | <Condition> <connective> <Condition>
<Item> ::= Name
<Cause> ::= <Atomic Sentence> | <expression> | <Cause> <connective> <Cause>
<connective> ::= AND | OR

Figure 4: Scenario/Use-case grammar

TITLE: CHECK BALANCE
GOAL: The User wants to check their account balance before withdrawing money.
CONTEXT:
PRE-CONDITION: User already logged onto the ATM
POST-CONDITION: Balance no longer displayed; ATM ready for a transaction.
ACTOR: User, Bank, ATM
RESOURCES:
EPISODES:
1. User selects balance of account.
2. User selects On Screen option.
3. ATM displays current balance on screen.
4. Bank retrieves User`s current balance from their account.
5. ATM prompts for new option.
ALTERNATE/EXCEPTION:
2a. User selects On Paper option.
2.1a ATM prints balance on receipt
2.2a User takes receipt

Figure 5: Check balance use case in ATM System [12]

(19) Check that every referenced scenario does not reference the main scenario [35];
(20) Check that the Title, Goal, Pre-condition or Episodes of a scenario is not already included in another scenario;
(21) Check that two scenarios does not have similar Titles;

2.2.3 Verify Consistency. A specification is consistent when two or more requirements are not in conflict with one another or with governing specifications and objectives [5]. The verification includes:

(1) Check the absence of non-determinism: simultaneously enabled operations [21];
(2) Check the absence of overflow: the number of elements in some resource exceeds a finite capacity;
(3) Check the absence of deadlocks [21];
(4) Check the absence of never enabled operations;
(5) Check Reversibility: error recovery is not possible [8].

2.3 C&L (Scenarios & Lexicons)

In order to automate the evaluation of scenarios quality, we developed a Java-based web application – C&L for editing, visualization and analysis of scenarios. C&L architecture is based on layers style, divided into modules and developed using *Domain-driven Design* practices. The modules group functionalities to manage users (*User*), projects (*Project*), lexicon symbols (Language used in the application - *LEL*), scenarios (*Scenario*) and to perform *Analysis* of structural and behavioral properties of scenarios. The input of the C&L is composed of projects containing scenarios in plain text format. The output is a set of formatted scenarios, where the relationships among scenarios are represented by hyperlinks (It facilitates the navigation between scenarios). Other modules include: (1) Tasks to *Pre-process* scenarios and annotate them with NLP information; and (2) A *Petri-Net* generator which uses mapping rules to translate scenarios into Petri-Nets. Fig. 6 shows the high-level architecture of the C&L.

2.3.1 Tools. Below, the tools behind our prototype tool:

NLP: An NLP pipeline is able to tokenize sentences into words; assign lemmas and *Part-of-speech* (POS) tags to words; and detect the grammatical relationships (*dependency parsing*) between words in a sentence (*subject-verb-object structure - SVO*). The Stanford Core NLP [26] provides the most complete annotators.

Petri-Net: It is a mathematical modelling and analysis language for studying systems that are concurrent, distributed, parallel, nondeterministic or stochastic [27]. A Petri-Net (See Fig. 11) is composed of nodes that denote *places* or *transitions*. Nodes are linked

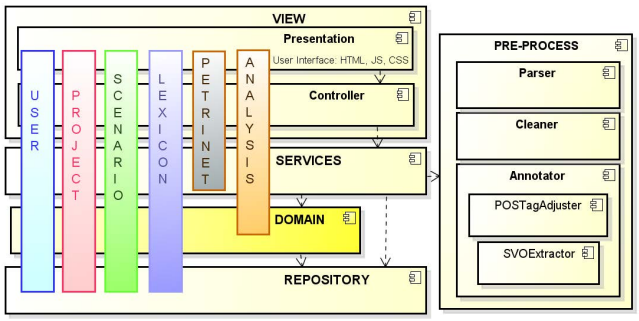

Figure 6: High-level architecture of the C&L.

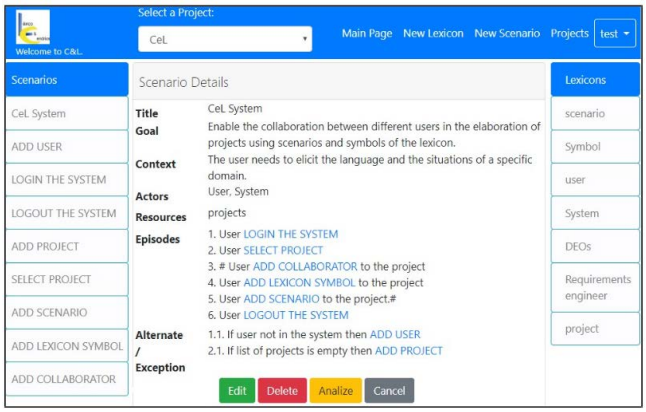

Figure 7: C&L functionalities.

The *scenario analysis* functionality is activated after the user selects a project or a scenario. This functionality generates a feedback containing a list of detected defects. As an example, Fig. 12 depicts an excerpt of the analysis feedback for the "ACCESS ATM" (Fig. 8) scenario of the "ATM System".

TITLE: ACCESS ATM
GOAL: User access the ATM.
CONTEXT: User wants to use the ATM.
PRE-CONDITION: ATM in ready state for new User
POST-CONDITION: User access granted; PIN and card validated.
ACTOR: User, ATM
RESOURCE:
EPISODES:
1. User inserts card into ATM.
2. ATM asks for a PIN.
3. User types in the numbers of his PIN and presses the Enter button
4. ATM asks for account type.
5. Customer selects account.
6. ATM displays User options.

ALTERNATE/EXCEPTION:
4a. ATM rejects unidentifiable car

Figure 8: Access ATM use case in ATM system [12]

3 ANALYSIS APPROACH

The approach implemented in the C&L tool takes as input scenarios, then, find defects that can be hidden within scenarios and their relationships with other scenarios. Requirements engineers CONSTRUCT scenarios, then, our approach, PRE-PROCESS, DISCOVER, DERIVE, ANALYZE and GENERATE a feedback (These activities are hidden and entirely accomplished by the C&L). Finally, Requirements engineers FIX scenarios with defects. The techniques behind these activities are explained in the following sub-sections.

Fig. 9 depicts the sequence of activities implemented in the C&L tool for automation of scenarios analysis.

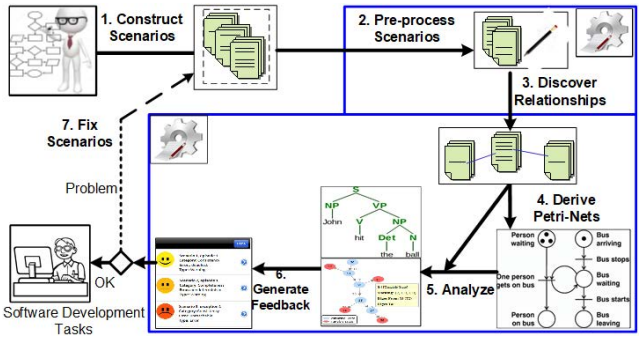

Figure 9: Overview of the analysis approach.

3.1 Construct

Requirements engineers construct scenarios for the different situations of the system (manual).

3.2 Pre-process

To be used for transformation and analysis, textual scenarios are:

Parsed: chunk textual descriptions into scenario sections, subcomponents and parts according to scenario model (Fig. 3) and grammar (Fig. 4).

Cleaned: to lowercase every character and remove irrelevant information and formatting symbols, such as empty lines, parenthesized texts, URLs, HTML tags, punctuation.

together by *arcs*. One important feature of Petri-Nets is the capability to analyze model properties through *reachability analysis*. Reachability analysis can reveal *overflows*, *non-deterministic* situations, *never enabled operations* and *deadlocks*. To this purpose, we modified the PIPE2 tool [15].

2.3.2 User Interface. The scenario of Fig. 7 provides an overview about the features of the C&L. The episodes of this scenario reference to scenarios that describe functionalities provided by the different modules that composes the system. The underlined terms are references to other scenarios (UPPERCASE) or to lexicon symbols (lowercase). Thus, the term *"SELECT PROJECT"* in Fig. 7 is a link to the scenario that describes how the user selects a project registered in the system. The term *"user"* is a link to a lexicon symbol. The concept of project is used within the system to represent different domains, where scenarios and lexicon symbols can be grouped.

Annotated: Each scenario statement is annotated with the corresponding *words, lemmas* and *POS tags*. Additionally, we analyze the *grammatical relations* (from dependency parsing) to highlight *nouns* and verbs in *subject, object* and *action-verb roles*.

Most of the NLP annotators perform their tasks with bad accuracy due to the imprecision of the POS tagging phase. Some of numerous causes are:

(1) The set of words used by POS tagger may not contain all the words occurring in the text being processed.
(2) More than one POS tag can be associated with a word, i.e., in English some words are both *noun* and *verb*, and a POS tagger can wrongly tag them as *noun*, *verb*, *preposition* or *adjective*. In fact, there are many words that can be used to name a person, place or thing and also describe an action. There are many *examples* of words that can be both nouns and verbs: "link", "step", "search", "contact", "download", "store", "delete", "use", "activate", "like", "form", "transfer", "view", "grant", "display", "broadcast", "order", "process", "bid", "prompt", "update", "access", "account", "release", so on.

For example, Fig. 10 shows that Stanford Core NLP POS tagger [26] wrongly annotated the *verbs* "types" and "presses" as *nouns*.

We improved the accuracy of parsing phase by adding a second phase with simple *rules to adjust* "Nouns", "Verbs", "Prepositions" and "Adjectives" based on dictionaries containing words are both "Noun" and "Verb". This strategy confirms that word within a sentence is a "Noun" or "Verb" or "Preposition" or "Adjective" based on neighbors POS tags. Implementation details are available on [7].

Fig. 10 shows how the words of a sentence are tokenized and tagged with POS tags like NN (noun) or VB (verb), then, typed *dependencies* (grammatical relations between individual words) are produced from a Parse Tree. Finally, we extract *subject, object* and *action-verb* from some *typed dependencies*.

3.3 Discover

A *scenario* is related to another scenario by:

- **Sequential relationship:** It occurs when we include the title (UPPERCASE) of a scenario (S_j) within a *pre-condition*, *post-condition*, episode sentence (*sub-scenario*) or *alternative* solution of another scenario (S_i). Thus, a statement of S_i will be expressed in S_j.
- **Non-Sequential relationship:** It might occur when two scenarios (S_i and S_j) have a high (>0.5) *Proximity Index* [22], i.e., they involve the participation of common actors, they access shared resources or they are executed in the same context (same pre-conditions or temporal location). Then, they might interact concurrently by *non-determinism* (S_i.pre-conditions = S_j.pre-conditions) and *synchronization* (S_i.pre-conditions $\cap$ S_j.post-conditions $\neq \emptyset$ AND S_j.pre-conditions $\cap$ S_i.post-conditions $\neq \emptyset$).

The scenario relationships are detailed in [33].

3.4 Derive

A *Petri-Net* ***PN*** can be derived from a scenario **S** by:

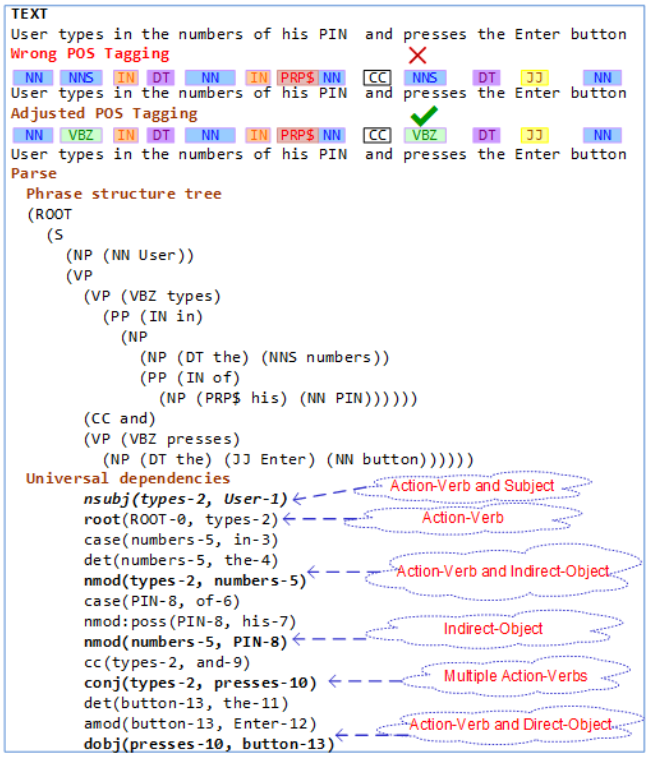

Figure 10: POS tagging and dependency parsing.

Identifying *event* occurrences (episode sentence and alternative solution step) and their guard *conditions* (condition, cause, pre-condition, post-condition and constraint);

Mapping *events* into Petri-Net *transitions*, and *conditions* into *PN input places* and *output places* (post-condition) of *transitions*.

Linking *PN transitions* and *places* by *arcs*, respecting the precedence order of *events*.

Scenarios are related (sequentially and non-sequentially) to other scenarios, then, in order to simulate the behaviour of a set of related scenarios we need to integrate related scenarios into a consistent whole Petri-Net. The first proposals for transforming and integrating related scenarios into Petri-Nets are detailed in [32] [33]. Fig. 11 depicts the Petri-Net from "ACCESS ATM" scenario.

3.5 Analyze

This activity implements the *verification heuristics* for searching indicators of violation of *unambiguity, completeness* and *consistency* properties. These heuristics evaluate structural (Static analysis) properties of scenarios and behavioral properties (Dynamic analysis) of equivalent Petri-Nets. Unexpected results are reported as defects, and defects in Petri-Nets are traced to defects in scenarios.

3.5.1 Static Analysis . It is useful for detecting unambiguity and completeness defect indicators.

We use lexical strategies like:

(1) *Dictionaries of frequently used ambiguous words:* check that scenario statements are free of ambiguity;

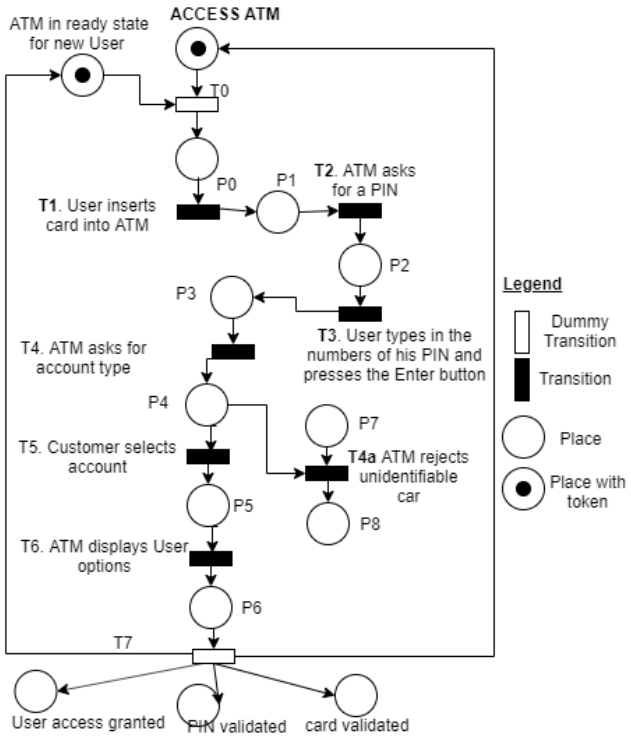

Figure 11: Petri-Net derived from "ACCESS ATM" scenario.

(2) *String Searching:* (a) verify that actors and resources are used in episodes or alternatives, (b) verify the existence of referenced scenarios
(3) *Regular Expressions:* check that scenario statements are conform to scenario model or grammar;
(4) *Levenshtein's distance:* compare pre-conditions, post-conditions, actors or resources of any two scenarios;
(5) *POS tagging:* (a) search for ambiguous words (implicit and subjective), (b) check the correct verb tense.

We use syntactic strategies like:

(1) *Dependency parsing:* (a) check that every scenario statement contains significant information like *action-verb, object* and *subject*, (b) check that every scenario statement does not contains multiple *action-verbs* and *subjects*, (c) check that every subject mentioned in episodes/alternatives are an Actor/resource/System, (d) check repeated episodes [23];
(2) *Syntactical similarity* (compare action-verbs and objects): (a) check that the Title describes the Goal, (b) check that titles and goals of scenarios are unique;

3.5.2 Dynamic Analysis. Static analysis is not useful for detecting consistency defects [14]; thus, we use *dynamic analysis* for simulating the behaviour of a set of Petri-Nets (derived from related scenarios) and to detect inconsistency or incorrectness indicators due to *non-determinism* and *synchronization* issues, we have made use of Place-Transition Petri-Nets [27] and *reachability analysis* strategies to detect *overflowed* resources, *deadlocks* and *never enabled* operations. Fig. 12 shows the defects pointed out by the C&L in the "ACCESS ATM" (Fig. 8) scenario. One important task of the feedback generator modules is to trace the defects reported from the Petri-Net to the source of these in the scenarios (sequence of episodes and alternatives involving related scenarios).

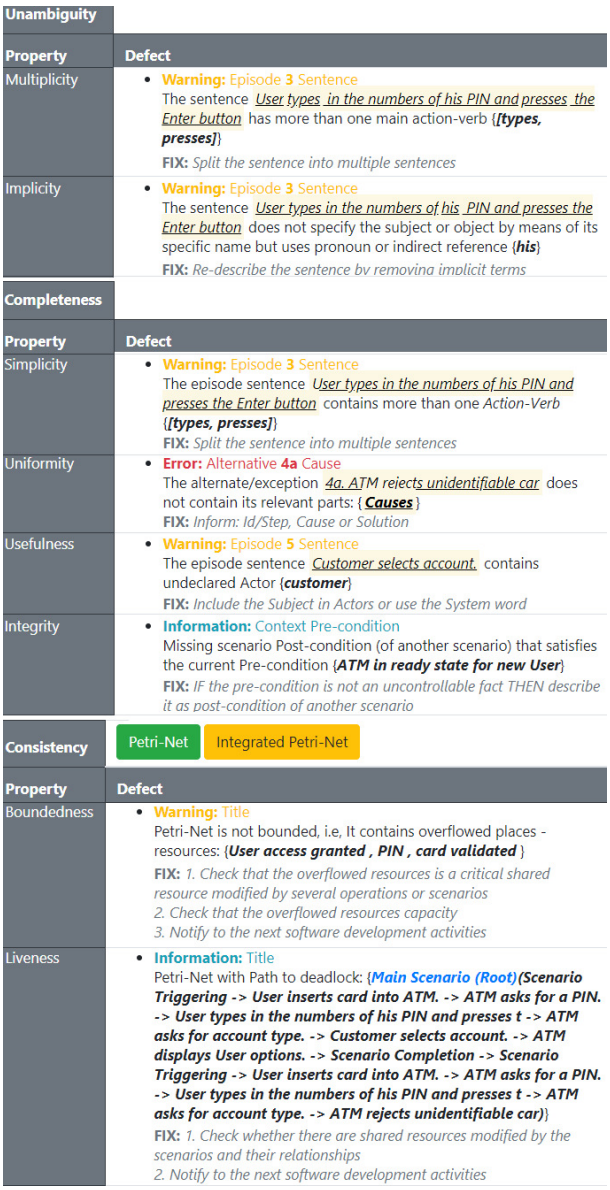

Figure 12: C&L Scenario analysis functionality.

3.6 Generate Feedback

Defects are reported following this format: *<Property> <Defect Category>: <Scenario Element> <Indicator> <Fix>*, where: *"Property"* is the quality violated, *"Scenario Element"* is the section or scenario statement where the defect occurs, *"Indicator"* gives a description of the defect for fixing, *"Fix"* gives a general recommendation for refactoring.

3.7 Fix

When defects are found, given the detailed information provided by the C&L, requirements engineers can review scenario descriptions and deal with defects via refactoring of scenarios.

4 EVALUATION OF THE C&L

We have taken the elicited requirements document of 4 projects (Table 1) as case studies. Somé [38] publicizes the material. Cox et al. [12], Ciemniewska and Jurkiewicz [10] make available the material analyzed and their preliminary analysis results (defects).

In order to compare the performance of the C&L, we need to contrast the *results* with another *baseline* solution. Graduate students (*volunteers* with background in use-cases/scenarios and analysis) of Computer Science manually constructed baseline solutions or validated existing preliminary analysis results. The case studies and solutions are available on [7].

Table 1: Characteristics of the Case Studies

	Broker System [38] (6 use cases)	ATM [12] (5 use cases)	DLibra [10] (15 use cases)	Mobile News [10] (15 use cases)
#episodes	32	33	80	89
#alternatives	9	5	26	5
#condition/Cause/Pre-cond.	15	8	33	0
#post-condition	2	9	0	0
Total (length)	58	55	139	94

We **evaluate** the accuracy of verification heuristics by:

(1) *Applying* the C&L tool to detect defects into projects;
(2) *Evaluating* whether detected defects are real defects by looking at the baselines and expert decisions;
(3) *Measuring* the accuracy of the analysis results using the evaluation metrics (***recall*** and ***precision***);

Unambiguity Results: Overall, the C&L achieved reasonable results (Table 2) with perfect recall and above 78% precision.

Table 2: Analysis of Unambiguity

	Broker System		ATM System		DLibra		MobileNews	
	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Vagueness	0	0	-	-	1	1	1	1
Subjectiveness	0	0	-	-	-	-	1	1
Optionality	-	-	-	-	-	-		
Weakness	-	-	-	-	1	1		
Multiplicity	1	1	1	1	1	1	1	1
Implicitly	1	1	1	1	1	1	1	0.81
Quantifiability	-	-	-	-	1	1		
Total	**1**	**0.78**	**1**	**1**	**1**	**1**	**1**	**0.95**

Completeness Results: Overall, the C&L produced reasonable results (Table 3) with perfect recall and above 73% precision.

Table 3: Analysis of Completeness

	Broker System		ATM System		DLibra		MobileNews	
	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Atomicity	1	1	-	-	1	0.5	0	0
Simplicity	1	1	1	0.83	1	1	1	1
Uniformity	1	1	1	1	1	1	1	1
Usefulness	1	1	1	1	1	1	1	1
Conceptually Soundness	1	1	0	0	-	-	0	0
Integrity	-	-	-	-	-	-		
Coherency	-	-	1	1	-	-		
Uniqueness	1	1	-	-	-	-		
Total	**1**	**1**	**1**	**0.82**	**1**	**0.97**	**1**	**0.95**

Consistency: It is difficult to elaborate referential solutions manually, then, it was difficult to compare the C&L results. However, The C&L simulated the Petri-Nets and detected situations involving *never enabled operations* and *deadlocks* (after handling exceptions).

Discussion: The C&L tool were precise, making only a few mistakes. The overall precision was quite high (93.5% precision and perfect recall). The results show promising results that indicate high potential for successful further improvements.

Although these set of scenarios describe some abstract systems, they show the typical defects of the industrial requirements specifications [10]. However more empirical experimentation with other projects and scenarios is advisable.

5 RELATED WORK

Many researchers have begun to see potential benefits from adding Natural Language Processing and Petri-Nets capabilities to requirements analysis approaches.

Most of the efforts are targeted towards the automatic detection of ambiguity indicators in *high level requirements* [3] [4] [16] [17] [20] [41] [43] or *user stories* [25] using NLP techniques. A few works [3] [4] [25] search for indicators that impacts on completeness (mainly, conformance to requirements templates).

In the context of scenarios, research to detect syntactic defects in use case descriptions with the aid of NLP techniques are proposed in [10] [24] [34]. They measure the compliance (*content and style*) of scenarios with writing guideline suggestions. In order to improve the accuracy of the NLP parser, domain knowledge is needed to train the parser. A few approaches [23] [24] [28] checked properties involving a set of scenarios (not fully automated). A few work [10] [23] [24] detail the heuristics for automation.

In order to evaluate behavioral properties of scenarios, some researches present systematic procedures to convert scenarios into formal representations: Labeled Transition Systems [9] [35], Statecharts [14] and Petri-Nets [8] [21] [32] [38]. The resulting models can be analyzed using tools, ensuring mainly consistency and correctness properties.

To facilitate the transformation, scenarios/use-cases are described using several semi-formal notations and intermediate models are created [21]. Relationships among scenarios/use-cases are partially considered [2] [21] [32]. Only [32] traces defects detected in executable models to scenarios. Related work does not make available automated tools. These shortcomings significantly hinder their applicability.

On the other hand, our approach: (1) supports most of the proposed scenario/use-case templates based on [13] or [23]; (2) provides powerful characteristics to deal with modularity, i.e., considers relationships among scenarios; and (3) implements fully automated transformation rules.

6 CONCLUSION

We work towards a *tool-supported* approach that checks acquired scenarios and their relationships by searching for indicators (e.g. missing information, wrong information and erroneous situations) that provide evidence of violation of *unambiguity*, *completeness*, and *consistency*. The C&L provides the following benefits: (1) it implements verification heuristics based on Dictionaries, Regular

Expressions, String Similarity, NLP and Petri-Nets; (2) it supports the most of the existing templates for describing scenarios or use cases; and (3) it achieves a high accuracy (the overall precision was 93.5% and the recalls were perfect). However, further empirical research is needed to corroborate these assumptions.

As such, our approach contributes to keep the software development within budget and delivery time.

Limitation: The transformation procedure from scenarios into Petri-Nets is sensitive to the correct syntax of scenarios. This mean that the transformation works well if a requirements engineer can write scenarios putting correct keywords (IF-THEN, DO-WHILE, so on) on statements. We assume that the use of keywords is well accepted by the stakeholders in the requirements process.

Future work includes: (1) investigating other heuristics to check other properties like traceability, modularity and testability; (2) improving the heuristics by including support tools for semantic analysis; (3) supporting other languages like Spanish and Portuguese.

ACKNOWLEDGMENTS

This work was supported by the Universidad Nacional de San Agustín de Arequipa (Project N°. IBAIB-06-2019-UNSA).

REFERENCES

[1] Camille B. Achour, Colette Rolland, Neil A. Maiden, and Carine Souveyet. 1999. Guiding use case authoring results of an empirical study. In *Proceedings of the IEEE International Symposium on requirements Engineering*. 36–43.

[2] Bende Anda, Kai Hansen, and Gunhild Sand. 2009. An investigation of use case quality in a large safety-critical software development project. *Information and Software Technology* 51 (2009), 1699–1711. Issue 12.

[3] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. 2015. Automated checking of conformance to requirements templates using natural language processing. *IEEE Transactions on Software Engineering* 41, 10 (2015), 944–968. https://doi.org/10.1109/TSE.2015.2428709

[4] Frederik S. Bäumer and Michaela Geierhos. 2018. Flexible ambiguity resolution and incompleteness detection in requirements descriptions via and indicator-based configuration of text analysis pipelines. In *Proceedings of the 51st Hawaii International Conference on System Sciences*. 5746–5755.

[5] Barry W. Boehm. 1979. Guidelines for verifying and validating software requirements and design specifications. In *Proceedings of the European Conference Applied Information Technology*. 711–719.

[6] Barry W. Boehm and Philip N. Papaccio. 1988. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* 14, 10 (1988), 1462–1477. https://doi.org/10.1109/32.6191

[7] C & L. 2019. Scenarios & Lexicons. https://github.com/edgarsc22/WACeL-Java.

[8] K. S. Cheung, T. Y. Cheung, and K. O. Chow. 2006. A petri-net-based synthesis methodology for use-case-driven system design. *Journal of Systems and Software* 79, 6 (2006), 772–790. https://doi.org/10.1016/j.jss.2005.06.018

[9] Minh-Hue Chu, Duc-Hanh Dang, Ngoc-Binh Nguyen, Minh-Duc Le, and Thi-Hanh Nguyen. 2017. Towards Precise Specification of Use Case for Model-driven Development. In *Proceedings of the Eighth International Symposium on Information and Communication Technology (SoICT 2017)*. ACM, 401–408.

[10] Alicja Ciemniewska and Jakub Jurkiewicz. 2007. *Automatic detection off defects in use cases*. Master's thesis. Poznan University of Technology, Faculty of Computer Science and Management, Institute of Computer Science.

[11] Alistair Cockburn. 2000. *Writing effective use cases* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Karl Cox, Aybuke Aurum, and Ross Jeffery. 2003. A use case description inspection experiment. In *Proceedings of the Australian workshop on software requirements*.

[13] Karl Cox and Keith Phalp. 2000. Replicating the Crews use case authoring guidelines experiment. *Empirical Software Engineering* 5, 3 (2000), 245–267.

[14] Christian Denger, Barbara Paech, and Bernd Freimut. 2005. Achieving high quality of use-ase-based requirements. *Informatik - Forschung und Entwicklung* 20, 1–2 (2005), 11–23. https://doi.org/10.1007/s00450-005-0198-4

[15] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. 2009. PIPE2: A tool for the performance evaluation of generalised stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review* 36, 4 (01 2009), 34–39.

[16] Henning Femmer, Daniel Mendez Fernandez, Stefan Wagner, and Sebastian Eder. 2017. Rapid quality assurance with Requirements Smells. *Journal of Systems and Software* 123 (2017), 190–213. https://doi.org/10.1016/j.jss.2016.02.047

[17] Alessio Ferrari, Gloria Gori, Benedetta Rosadini, Iacopo Trotta, Stefano Bacherini, Alessandro Fantechi, and Stefania Gnesi. 2018. Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. *Empirical Software Engineering* 23, 6 (2018), 3684–3733.

[18] Donald Firesmith. 2007. Common Requirements Problems, their negative consequences and the Industry best practices to help solve them. *Journal of Object Technology* 6, 1 (2007), 17–33.

[19] Martin Glinz. 2000. Improving the quality of requirements with scenarios. In *Proceedings of the Second World Congress for Software Quality*. Yokohama, 55–60.

[20] Giuseppe Lami, Stefania Gnesi, Fabrizio Fabbrini, Mario Fusani, and Gianluca Trentanni. 2004. An Automatic Tool for the Analysis of Nature Languaje Requirements.

[21] Woo J. Lee, Yong R. Kwon, and Sung D. Cha. 1998. Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. *IEEE Transactions on Software Engineering* 24, 12 (1998), 1115–1130.

[22] Julio C. S. P. Leite, Jorge H. Doorn, Graciela D. Hadad, and Gladys N. Kaplan. 2005. Scenario inspections. *Requirements Engineering* 10, 1 (2005), 1–21.

[23] Julio C. S. P. Leite, Graciela D. Hadad, Jorge H. Doorn, and Gladys N. Kaplan. 2000. A Scenario construction process. *Requirements Engineering Journal* 5, 1 (2000), 38–61. https://doi.org/10.1007/s00766-003-0186-9

[24] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. 2014. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE International Conference on Automate Software Engineering (ASE '14)*. ACM, New York, NY, USA, 785–790.

[25] Garm Lucassen, Fabiano Dalpiaz, Jan M. E. M. van der Werf, and Sjaak Brinkkemper. 2015. Forging high-quality user stories: Towards a discipline for agile requirements. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*. 126–135. https://doi.org/10.1109/RE.2015.7320415

[26] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Standford Core NLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demostrations*. Association for Computational Linguistics, Baltimore, Maryland, 55–60. https://doi.org/10.3115/v1/p14-5

[27] Tadao Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580. https://doi.org/10.1109/5.24143

[28] Keith T. Phalp, Jonathan Vincent, and Karl Cox. 2007. Assessing the quality of use case descriptions. *Software Quality Journal* 15, 1 (2007), 69–97.

[29] PMI. 2014. Requirements Management: Core Competency for Project and Program Success. In-Depth Report. Internet.

[30] Edgar Sarmiento. 2016. *Analysis of Natural Language Scenarios*. D. Sc. Thesis. PUC-Rio.

[31] Edgar Sarmiento. 2019. *Construcción de una Herramienta para el Análisis de Requisitos de Software Descritos en Lenguaje Natural*. Bachelor's Thesis. UNSA.

[32] Edgar Sarmiento, Julio C. S. P. Leite, and Eduardo Almentero. 2015. Analysis of scenarios with Petri-Net models. In *2015 29th Brazilian Symposium on Software Engineering (SBES)*. Belo Horizonte, 90–99.

[33] Edgar Sarmiento, Julio C. S. P. Leite, Eduardo Almentero, and Guina S. Alzamora. 2016. Test scenario generation from natural language requirements descriptions based on Petri-Nets. *Electronic Notes in Theoretical Computer Science* 329 (2016), 123–148. https://doi.org/10.1016/j.entcs.2016.12.008.

[34] Avik Sinha, Stanley M Sutton, and Amit Paradkar. 2010. Text2Test: Automated inspection of natural language use cases. In *Third International Conference on Software Testing Verification and Validation (ICST)*. 155–164.

[35] Daniel Sinning, Patrice Chalin, and Ferhat Khendek. 2009. LTS semantics for use case models. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. ACM, New York, NY, USA, 365–370.

[36] Fabio L. Siqueira and Paulo S. M. Silva. 2011. An essential textual use case meta-model based on an analysis of existing proposals. In *Anais do Workshop em Engenharia de Requisitos (WER11)*.

[37] IEEE Computer Society. 1998. IEEE Recommended practice for software requirements specifications.

[38] Stephane S. Somé. 2010. Formalization of textual use cases based on Petri Nets. *International Journal of Software Engineering and Knowledge Engineering* 20 (2010), 695–737. Issue 5.

[39] The Standish Group. 2016. Chaos Report. Internet. https://www.standishgroup.com/outline

[40] Saurabh Tiwari and Atul Gupta. 2015. A sistematic literature review of use case specification research. *Information and Software Technology* 67 (2015), 128–158.

[41] Sri F. Tjong. 2008. *Avoiding ambiguity in requirements specifications*. D. Sc. Thesis. University of Nottingham Malaysia Campus.

[42] Willem Van Galen. 2012. Use Case Goals.

[43] William M. Wilson, Linda H. Rosernberg, and Lawrence E. Hyatt. 1997. Automated Analysis of Requirement Specifications. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*.

[44] Tao Yue, Lionel C. Briand, and Yvan Labiche. 2013. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22, Article 5 (2013), 38 pages. Issue 1. https://doi.org/10.1145/2430536.2430539