The Journal of Systems and Software 208 (2024) 111901

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Early analysis of requirements using NLP and Petri-nets ☆

Edgar Sarmiento-Calisaya [a,*], Julio Cesar Sampaio do Prado Leite [b]

[a] Universidad Nacional de San Agustín de Arequipa, Arequipa, Peru
[b] Instituto de Computação, Universidade Federal da Bahia, Salvador, Brazil

ARTICLE INFO

Dataset link: https://github.com/edgarsc22/WACeL-Java

Keywords:
Analysis
Verification
Scenario
Use case
Natural language processing
Petri-nets

ABSTRACT

Scenario-based approaches are widely used for software requirements specification. Since scenarios are usually written using natural language, specifications may have statements that are ambiguous, unnecessarily complicated, missing, duplicated, or conflicting. Requirements quality is challenging since it is hard to achieve consistency in requirements products. Unfortunately, if done manually, analysis of textual scenarios can be an arduous, time-consuming, and error-prone activity. This work rethinks the unambiguity, completeness, consistency, and correctness properties of scenario-based specifications; and how static and dynamic analysis strategies could automatically evaluate them. To do so, we introduce an automated requirements analysis approach to check both structural and behavioral aspects of scenarios, which combines natural language processing, Petri-nets, and visualization techniques for: (i) identifying certain types of defects and their indicators; (ii) highlighting scenario statements or relationships among scenarios that can lead to defects; and (iii) foreseeing scenario execution paths that can lead to inconsistencies. We show the feasibility of the proposed approach through the analysis of four projects specified as scenario-based descriptions. Overall, our approach produced reasonable results, with precision greater than 89% and recall greater than 98%. Our work allows researchers, as well as practitioners, to improve the quality of scenarios through an automated analysis approach.

1. Introduction

Reports about software projects performance in industry (Firesmith, 2007; PMI, 2014; The Standish Group, 2016) show that *requirements analysis* activities are closely related to superior quality software. However, *verification and validation* are usually performed through a manual procedure for reading software requirements specification (SRS) documents, finding defects, and addressing these problems (Condori-Fernandez et al., 2014). In addition, requirements engineers tend to detect simple defects that are easier to detect – *linguistic defects* (Ciemniewska and Jurkiewicz, 2007; Lami et al., 2004; Phalp et al., 2007), and leave aside defects that are harder to detect – *defects arising from relationships among requirements*. Thus, proper analysis is a complex process, which requires significant effort and takes time.

Natural language (NL) scenario descriptions, is one of the widely used techniques for eliciting and specifying functional requirements in industry (Tiwari and Gupta, 2020; Wang et al., 2020). These techniques help to better understand the interactions between *actors* and a *system* (Cockburn, 2000; Jacobson et al., 1992), focusing on the *behavior* of an application (do Prado Leite et al., 2000). The most prominent languages to write scenarios are *use case* descriptions (Cockburn, 2000; Jacobson et al., 1992), *scenario* descriptions (do Prado Leite et al., 2000), or their variations. However, they lack precise semantics to: uncover the relationships among scenarios, support their early analysis, and highlight the defects.

Given that an SRS is the basis for software construction, ensuring its quality is crucial, i.e., the early identification of defects and their removal can greatly improve the quality of the artifacts generated from them. To this purpose, several approaches impose keywords and restriction rules on the expressiveness of requirements or use linguistic techniques to address aspects related to unambiguity and completeness, but not sufficient to address aspects related to consistency.

To illustrate this challenge and show the usefulness of the proposed approach, new defects were introduced in the "*Submit Order*" scenario (as in Fig. 1a) of the "*Broker System*" project (Appendix E).[1] In Fig. 1a, words in red stand for defect indicators, and the capitalized phrases represent the title of the included scenarios (e.g., PROCESS BIDS). This scenario representation shares similarities with others in the literature (Section 2.1).

☆ Editor: Dr. P. Lago.

* Corresponding author.
E-mail addresses: esarmientoca@unsa.edu.pe (E. Sarmiento-Calisaya), julioleite@ufba.br (J.C.S. do Prado Leite).

[1] Appendix E is available at https://bit.ly/3dDzk5x.

https://doi.org/10.1016/j.jss.2023.111901
Received 16 April 2023; Received in revised form 1 November 2023; Accepted 6 November 2023
Available online 17 November 2023

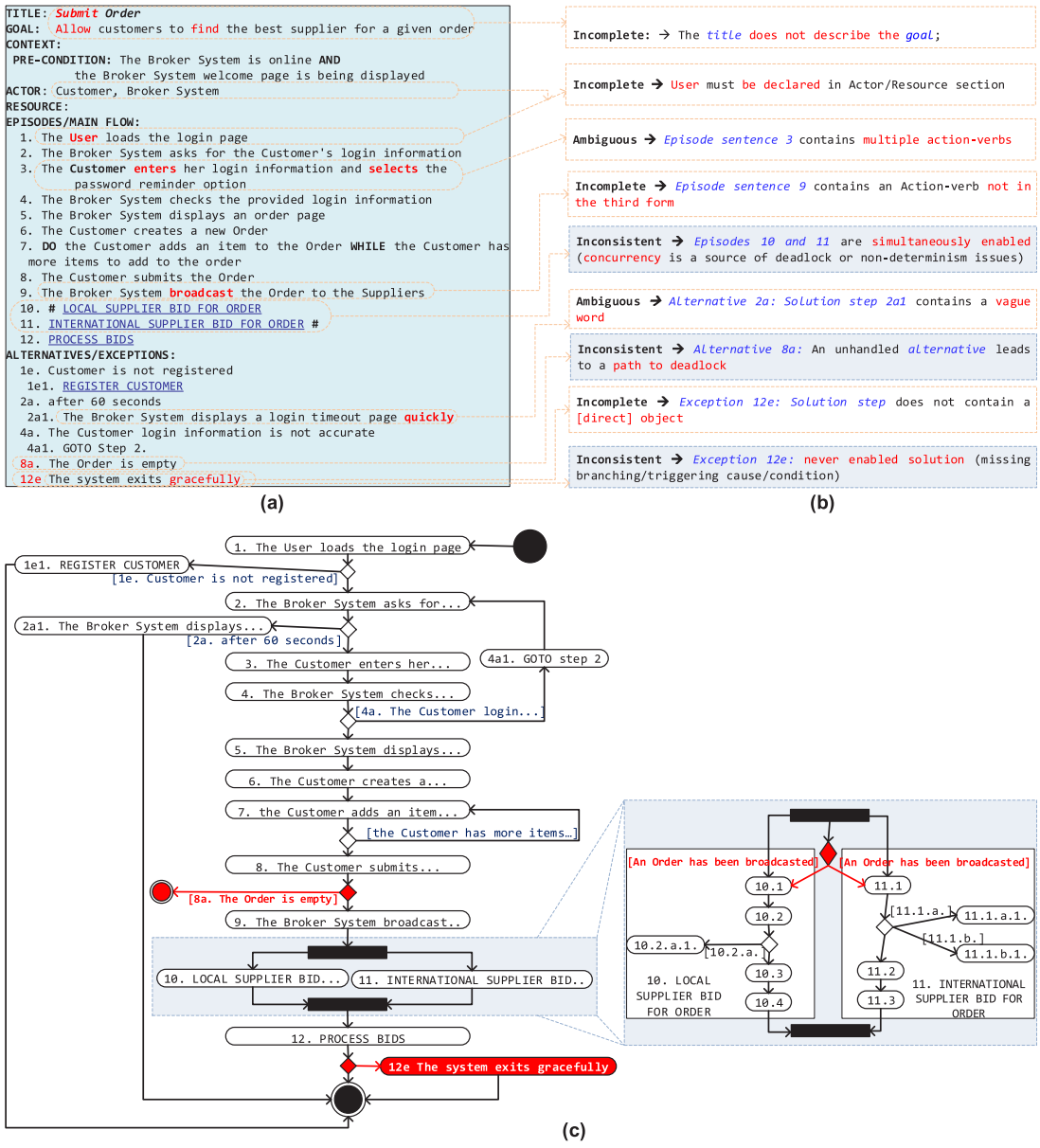


Fig. 1. A defective Submit Order scenario (a), its potential defect indicators (b), and its visual representation (c).

Fig. 1b highlights the statements involved in potential defects within the "*Submit Order*" scenario (Fig. 1a). This scenario is described by statements that have: *vague terms, multiple action verbs, undeclared actors, action verbs in the incorrect form, missing objects, never enabled actions, simultaneously enabled actions, and a path to deadlock*.

Fig. 1c illustrates the "*Submit Order*" scenario and its included scenarios (Local and International Suppliers) through a visual representation – *UML activity diagram*, from which it is possible to identify the occurrence of possible inconsistencies. The symbol # signals parallel or concurrent episodes or indistinct sequential order. A *non-deterministic situation* occurs when the resource "An order has been broadcasted" can be allocated in either activities "10.1" or "11.1"; thus, there is not an explicit decision on which activity would have the resource. A *path to deadlock* occurs when the condition "The order is empty" is true, so activity "9" will never be instantiated because it requires the completion of activity "8", thus, the scenario will never terminate successfully. A *never enabled operation* occurs when activity "12e" can be never instantiated because it requires an explicit decision. The red edges and nodes represent the trace to possible defects.

Visualization or simulation in requirements engineering and software engineering activities have been widely discussed to support requirements inconsistencies management (Denger et al., 2005; Abad et al., 2016); bridge the gap between specifications, design, and development artifacts (Yue et al., 2013; Tiwari et al., 2020); and automate test generation (Wang et al., 2020). On the other hand, research conducted by Hinchey et al. (2008) Weyns et al. (2012), Denaro and Pezze (2004), He (2013), Domı et al. (2012) reported the use of formal representations in software engineering activities.

UML-based approaches (Gutiérrez et al., 2008; Yue et al., 2010) for scenarios visualization and analysis were proposed; however, UML does not provide execution semantics. Petri-nets (Murata, 1989) and Statecharts (Harel, 1987) can take advantage of the execution semantics of formal languages to simulate the behavior of a set of scenarios, detect inconsistencies, and mark the paths that lead to them.

If done manually, requirements analysis can be time-consuming, tedious, and error-prone (Pohl and Rupp, 2015). Particularly, the activity presents a challenge when requirements interact among them; and requirements engineers cannot foresee the possible inconsistencies, unless they simulate the related requirements. As such, this work is motivated

by the need of solutions to check both *structural* and *behavioral* aspects of scenario descriptions *without* reliance on formal semantics. To this end, we make the following four contributions:

- A conceptual *model for scenarios*, which considers the most frequent concepts/elements of existing templates and allows the discovery of complex relationships among scenarios.
- A *quality model* for scenarios, which models the impacts between *unambiguity*, *completeness*, *consistency*, and *correctness*; as well as the properties and verification heuristics associated with these qualities.
- An *automated analysis approach*, which takes as input textual scenarios and evaluates their structural and behavioral aspects using *natural language processing* and *Petri-nets*-based strategies. As a result, it uncovers potential *defects* that can be hidden within scenarios, as well as in their relationships.
- A *Petri-net* and visualization-based strategy for early validation of scenarios, i.e., it creates visual representations about scenarios behavior to foresee scenario execution paths that can lead to inconsistencies, when implemented.

To evaluate our automated analysis approach, we conducted a multi-case study with four projects specified as scenario-based descriptions. The evaluation addresses questions related to: (1) whether the proposed approach detects defects in SRSs in due time, and (2) whether the proposed approach detects defects in SRSs accurately. Overall, the proposed analysis approach achieved promising results.

This work is organized as follows: Section 2 presents the background; Section 3 defines the scenario model and quality properties; Section 4 details the analysis approach; Section 5 evaluates the analysis approach; Section 6 presents the related work and Section 7 concludes the work.

2. Background

This section introduces the scenarios templates, natural language processing, Petri-nets, and the C&L prototype tool.

2.1. Scenarios

Cockburn (2000) and Glinz (2000) define scenarios as "an ordered set of interactions between a system and a set of actors external to the system – a use case description". do Prado Leite et al. (2000) define scenario as "...describe situations in the macrosystem and not only a sequence of interactions between a user and a system". They are widely used in requirements engineering, as they help developers and other stakeholders to better understand the requirements and their interface with the environment, i.e., scenarios are easier to read and write.

There is a wide variety of *templates* for representing scenarios in the literature such as use case descriptions (Cockburn, 2000; Jacobson et al., 1992), scenario descriptions (do Prado Leite et al., 2000), or their variations; each one with quite different purposes. Therefore, scenarios can take many forms and supply distinct types of information on different levels of abstraction and formalism.

Most of the studies (do Prado Leite et al., 2000; Siqueira and Silva, 2011; Tiwari and Gupta, 2015, 2020) about templates reveal that these emphasize on specifying the *name* (or title), *goal* (or description), *pre-condition*, *post-condition*, *actor*, *main flow*, and *alternative/exception flow* sections. These templates make use of a single or two-column format to specify the plain textual descriptions. A detailed analysis of nine different variants of scenario/use-case templates (Jacobson et al., 1992; Cockburn, 2000; do Prado Leite et al., 2000; Insfrán et al., 2002; Kruchten, 2003; Somé, 2010; Sinha et al., 2010; Yue et al., 2013; Chu et al., 2017) resulted in the identification of eight significant sections (as in Table 1), which are described in Fig. 2.

Other approaches (Chu et al., 2017; do Prado Leite et al., 2000; Sinha et al., 2010) recommend the use of a section to hold the data or

```
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
                   <simple sentences or name of other-scenarios linked by AND
                   or OR>
 POST-CONDITION:   What should be true after the basic flow ends
                   <simple sentences or name of other-scenarios linked by AND
                   or OR>
ACTOR:             Person, another system, device, component or organization
                   structure directly involved with the situation
                    <actor1>, <actor 2>, ... <actorn>
[RESOURCE:         Physical element or data used to achieve its goal]
                    <resource1>, <resource2>, ... <resourcen>
EPISODES/          The most common and successful flow of events. An event is
                   described using a simple, conditional, loop or
    MAIN FLOW:     parallel (or concurrent or indistinct order) structure,
                   and special keywords. Conditions are linked by AND or OR.
                    <Step> <Action statement or name of other-scenario>
                    1 <Sentence>
ALTERNATIVES /     Describes a deviation (branch) in another flow of events.
    EXCEPTIONS:    An alternative flow depends on a condition occurring in a
                   specific step of the main flow and is composed of a
                   sequence of numbered sentences. Conditions are linked by
                   AND or OR. It can be an:
                   - Alternate: less-frequent flow to achieve the goal.
                   - Exception: flow that leads to an obstacle to achieve the
                     goal.
                     <Step><Ref> <Condition causing branching>[:]
                      [<Step><Ref><Step>] <Action or name of other-scenario>
                     1.a <Cause>:
                      1.a.1 <Solution Step Sentence>
                     2E <Cause>:
                      2E1  <Solution Step Sentence>
```

Fig. 2. Common sections in scenario/use-case templates.

physical elements during the execution of a scenario/use-case: *resource* or *business item*.

To facilitate the transformation into other software artifacts (e.g., domain models and test cases), some semi-formal templates suggest that each *alternative* flow should have *post-conditions* associated with it (Yue et al., 2013), and each event in the*main flow* should have *pre-conditions* and *post-conditions* associated with it (Chu et al., 2017; Lee et al., 1998). Templates analyzed by Siqueira and Silva (2011), and Tiwari and Gupta (2015) suggest the use of the following types of *action statements* in the main flow: (1) *Conditional*, specifies conditions to execute an action; (2) *Loop*, specifies conditions to execute an action repeatedly while conditions are true; (3) *Parallel*, specifies the starting point of two or more actions executed in an indistinct, concurrent, or parallel order.

Tiwari and Gupta (2015) found that most templates recommend: (1) The use of a *numbered* style format in the main and alternative flows; (2) The use of some *keywords* such as IF-THEN, DO-UNTIL, RESUME or GO-TO; (3) That alternative numbering should match the numbers (steps) in the main flow; (4) That every alternative flow must return to some step of the main flow or finish the scenario.

The action statements within scenarios are often described following known writing guidelines (e.g., Chu et al., 2017; Cockburn, 2000; Cox and Phalp, 2000; Siqueira and Silva, 2011; Somé, 2010; Yue et al., 2013), i.e., by simple sentences centered on the main verb (subject + action-verb + object | prepositional-phrase). Fig. 1a describes the "*Submit Order*" scenario using the general template's sections described in Fig. 2.

2.2. Natural language processing – NLP

NLP is a range of computational techniques for the automatic analysis and representation of human language (Cambria and White, 2014). And, *Information Extraction* research (concerned with extracting information from unstructured text) is often the most widely used in the context of software engineering (Garousi et al., 2020). A typical NLP pipeline for information extraction is composed of the following tasks (details in Manning et al., 2014):

- *Tokenization*: tokenizes the text into words – *tokens*.
- *Segmentation*: splits tokens of text into sentences.

Table 1
Scenario/Use-case templates and their most common sections.

Section	Jacobson et al. (1992)	Cockburn (2000)	do Prado Leite et al. (2000)	Insfrán et al. (2002)	Kruchten (2003)	Somé (2010)	Sinha et al. (2010)	Yue et al. (2013)	Chu et al. (2017)
Name/Title	✓	✓	✓	✓	✓	✓	✓	✓	✓
Goal/Description	✓	✓	✓	✓	✓	✓		✓	✓
Pre-condition	✓	✓	✓		✓	✓		✓	✓
Post-condition	✓	✓	✓		✓	✓		✓	✓
Actor		✓	✓	✓	✓	✓	✓	✓	✓
Re-source/Business item			✓				✓		✓
Episodes/Main flow	✓	✓	✓	✓	✓	✓	✓	✓	✓
Alterna-tives/Exceptions	✓	✓	✓	✓	✓	✓	✓	✓	✓

- *Part of speech (POS) tagging*: labels tokens with their part-of-speech tag, e.g., noun (NN), verb (VB), etc. The Penn Treebank describes 48 tags (Marcus et al., 1993).
- *Morphological analysis*: reduces words to a common base/root form, i.e., generates lemmas for tokens.
- *Parsing*: recognizes a sentence and assigns a syntactic structure to it: (1) *Constituency parsing* breaks a text into sub-phrases and generates a phrase structure tree, where Non-terminals are types of phrases (noun or verb phrases), whereas the terminals are the words in the sentence; (2) *Dependency parsing* focuses on the grammatical relationships (typed dependencies are detailed in de Marneffe et al., 2014) between words in a sentence (e.g., the classic subject-verb-object structure); (3) *Shallow parsing* or *Chunking* breaks the sentence into groups (of words) containing sequential words of sentence, that belong to a noun group, verb group, etc.
- *Named entity recognition*: recognizes named, numerical and temporal entities.
- *Coreference resolution*: finds all expressions such as pronouns that refer to the same entity in a text.

NLP pipeline tasks are often interwoven, e.g., first applying POS tagging, and afterward conducting parsing.

Fig. 3 shows how the words of a sentence are tokenized and tagged with *POS tags*, then, *typed dependencies* are produced. The Stanford Core NLP (Manning et al., 2014) parser generates a *Typed Dependency* as a triplet structure *(td_name, gov, dep)*, where *td_name* represents the name of the dependency, *gov* represents the head word and *dep* is the dependent word.

Generally, NLP annotators perform their tasks with bad accuracy due to the imprecision of the POS tagging phase. Frequent causes are: (1) the POS tagger may not hold all the words occurring in the text being processed; (2) the POS tagger can incorrectly tag a word as a noun, verb, preposition or adjective. In fact, many words are both nouns and verbs, i.e., they can be used to name an entity and describe an action.

POS tagging and parsing outputs can be used to pinpoint potential ambiguous and incomplete requirement statements, e.g., a statement containing multiple action-verbs.

2.3. Petri-nets

Petri-net is a modeling and analysis language for studying systems that are concurrent, distributed, parallel, non-deterministic, or stochastic (Murata, 1989). A Petri-net (Fig. 4a) is composed of nodes that denote *places* or *transitions*. These nodes are linked by *arcs*. *Transitions* model the activities or events that can occur, thus changing the state of the system; transitions are only allowed to fire if they are enabled, which means that all the pre-conditions (input places) have been fulfilled. *Places* (placeholders for tokens) model communication channels, resources, buffers, geographical locations, or a possible state or condition; the current state of the system being modeled is called *marking*,

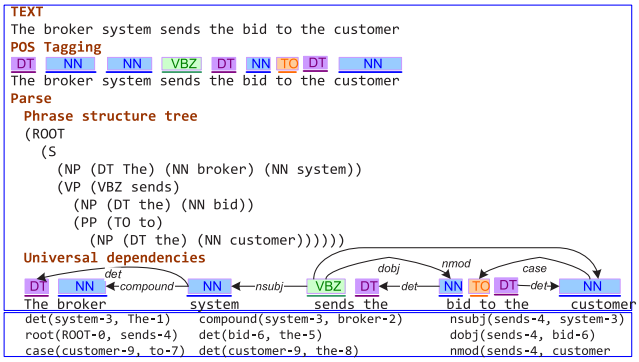


Fig. 3. NLP POS tagging and parsing example.

which is given by the number of tokens in each place. *Tokens* model physical or information objects, collections of objects, or indicators of state or condition. *Input arcs* start from places and end at transitions, and *output arcs* start at a transition and end at a place.

Definition 2.1. A ***Place-Transition Petri-Net*** is a five-tuple $PN = (P, T, F, W, M_0)$ where $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_m\}$ is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $W : F \to \{1, 2, \ldots\}$ is a weight function, $M_0 : P \to \{0, 1, 2, \ldots\}$ is the initial marking and $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. A PN is said to be ordinary — marked graph (Cheung et al., 2006) iff $W : F \to 1$.

Definition 2.2. For a PN , a ***marking*** is a function $M : P \to \{0, 1, 2, \ldots\}$, where $M(p)$ is the number of tokens in p. M_0 represents PN with an initial marking.

When a transition *fires*, it removes tokens from its input places and adds them to its output places. The number of tokens removed/added depends on the *weight* of each arc.

Definition 2.3. A transition t is ***enabled*** at a marking M if $M(p) \geq w(p, t)$ for any $p \in^\circ t$ where $^\circ t$ is the set of input places of t. On firing t, M is changed to M' such that $\forall p \in P : M'(p) = M(p) - W(p, t) + W(t, p)$. $M[t\rangle M'$ denotes firing t at marking M.

Definition 2.4. For a PN, a sequence of transitions $\delta = (t_1, t_2, \ldots, t_n)$ is called a ***firing sequence*** if and only if $M_0[t_1\rangle, [t_2\rangle, \ldots, [t_n\rangle M_n$. In notation, $M_0[PN, \delta\rangle M_n$ or $M_0[\delta\rangle M_n$.

Definition 2.5. For a PN , a marking M is said to be ***reachable*** if and only if there exists a firing sequence δ such that $M_0[\delta\rangle M$. In notation, $M_0[PN, *\rangle M$ or $M_0[*\rangle M$ represents the set of all reachable markings of PN.

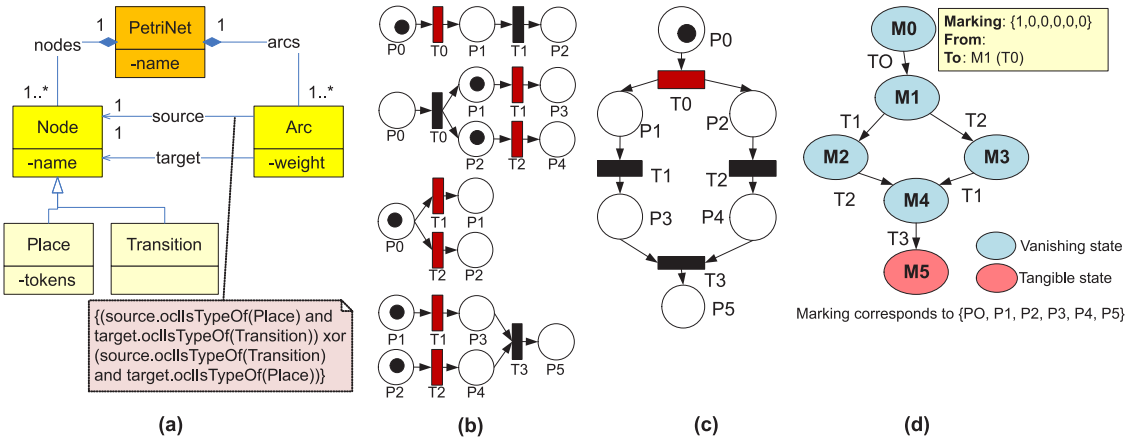


Fig. 4. Petri-net metamodel (a), modeling Petri-nets (b), Petri-net example (c) and its reachability graph (d).

In Petri-nets, a subset of transitions interact by the following relationships (as in Fig. 4b): (1) ***Sequential***, a transition *T0* fires before another *T1*; (2) ***Concurrency***, transitions *T1* and *T2* may fire simultaneously; (3) ***Non-determinism***, only one of transitions *T1* or *T2* may fire; and (4) ***Synchronization***, a transition *T3* fires after transitions *T1* and *T2* fire.

One important feature of Petri-nets is the capability to analyze model properties. The *reachability analysis* (Murata, 1989) generates a reachability graph that contains all possible *reachable markings*, i.e., states of the system over time. Dynamic properties like reachability, boundedness, liveness and reversibility are also evaluated. Fig. 4c shows a marked ($T0$ is enabled) Petri-Net; and Fig. 4d shows its reachability graph, where $M0$ is the initial marking labeled with $\{1,0,0,0,0,0\}$.

Definition 2.6 (*The Reachability Graph*)**.** of a *PN* is a directed graph $G = (N, A)$, where nodes N corresponds to reachable markings ($N = M_0[*\rangle M$) and arcs A correspond to feasible transitions ($A = T$). Thus, $Markings$ are states reached from the initial marking M_0 by firing transitions.

Definition 2.7 (*Reachability*)**.** is the problem of finding if $M_n \in M_0[*\rangle M$ for a given marking M_n in a *PN* with initial marking M_0.

Definition 2.8. A *PN* is ***bounded*** if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e., $M(p) \leqslant k$ for every place p and marking M. A *PN* is safe if it is $1 - bounded$.

Definition 2.9. For a *PN*, a transition t is said to be ***live*** if it is possible to ultimately fire it by progressing through some firing sequence, i.e., if and only if $\forall M \in [M_0\rangle, \exists M' : [*\rangle M'[t\rangle$. PN is said to be $live$ if and only if every transition is live – deadlock-free.

Definition 2.10. For a *PN*, if there exists a marking $M \in [M_0\rangle$ such that $\neg M[t\rangle$ for any $t \in T$, then marking M is called a $deadmarking$ of *PN*, i.e., a ***deadlock***. A *PN* is called $deadlock - free$ if deadlock does not exist in *PN*. $\neg M[t\rangle$ denotes that t is disenabled under M.

Definition 2.11. A *PN* is ***persistent*** if, for any enabled transitions, the firing of one transition will not disable the others. In parallel systems, persistence is closely related to the absence of ***non-determinism***.

Definition 2.12. A *PN* is ***reversible*** if M_0 is reachable from each other reachable marking M, i.e., $\forall M \in [M_0\rangle : M[*\rangle M_0$.

Petri-nets analysis can simulate the behavior of a set of related scenarios, support the early detection of inconsistencies, and mark the paths that lead to them.

2.4. Scenarios & Lexicons (C&L)

The C&L is a web application for editing, visualization, analysis, and simulation of scenarios. The input of the C&L is composed of projects containing scenarios in plain text format. The output is: (1) a set of formatted scenarios, where the relationships among them are represented by hyperlinks, (2) a report of analysis of structural and behavioral aspects of scenarios, and (3) a module for simulating equivalent Petri-nets. Details about its implementation are available in Sarmiento-Calisaya et al. (2020).

3. A model for scenarios

This section describes a language for describing scenarios, which supports the common sections of the general template described in Fig. 2; and the quality properties of scenarios and their verification heuristics.

3.1. Scenarios language

A *scenario* is a partial description of the application behavior that occurs in a specific *context*, with needed *resources* and *actors*. It must satisfy a *goal* that is achieved by performing *events* occurring in its *episodes* and *alternative* flows, each guarded by *conditions* or restricted by *constraints*.

An *event* can be an episode sentence or an alternative solution step, which describes a signal, a message, an action, or an interaction involving users, organizations, system components, resources, the system environment, and the system. A *condition* can be the state of an actor, a resource, or the system.

The *context* is described through at least one of these sub-comp onents: (1) *Pre-condition*, (2) *Post-condition*, (3) *Geographical Location* and (4) *Temporal location*.

Episodes describe the main flow of actions, and they are simple, optional, conditional, and loop ones. *Optional* episodes may or may not take place depending on conditions that cannot be detailed (It is like an optional feature that contributes to the goal). *Conditional/Loop* episodes depend on conditions and must be described using special keywords like IF-THEN, DO-UNTIL, or their variations. Optionally, an episode is carried out only when a set of *pre-conditions* is satisfied, and it generates effects (*post-conditions*) on some resource or the system.

A sequence of *episodes* implies a precedence order. A *non-sequential* (parallel, concurrent, or indistinct) order can be expressed using the #<Episode Series># structure. In Fig. 1a, episodes 10 and 11 are performed in parallel.

Alternative flows can arise during the execution of the episodes, which show that there exists either a less-frequent way to achieve the

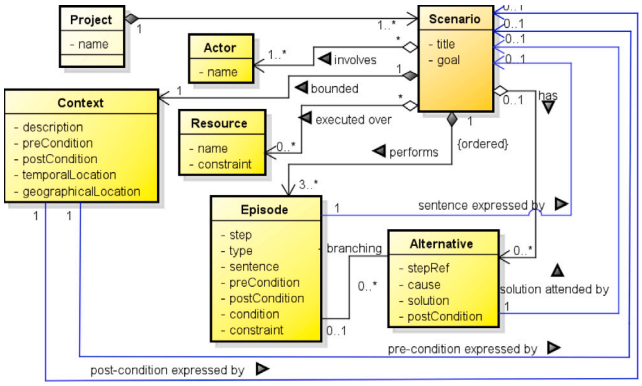


Fig. 5. Scenario conceptual model.

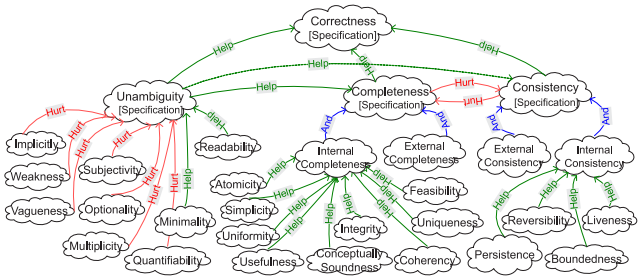


Fig. 6. Quality properties of scenarios and their impacts.

goal – *an alternate* or an obstacle to achieve the goal – *an exception*. An alternative always depends on a condition occurring in a specific episode and it is treated by a sequence of numbered solution steps. Every *alternate* must finish the scenario; or return to a specific episode and using associated keywords and fields "(GO | BACK | RETURN | RESUME) + TO + [STEP | EPISODE] + <Step>". An *exception* must finish the scenario or redirect to another scenario (catch). Optionally, an alternate flow has *post-conditions* associated with it.

A *constraint* restricts the quality (non-functional aspect) with which the goal is achieved, the resources are needed, and the episodes are performed. It is an attribute of a resource or an episode.

The ***scenario statements*** (title, goal, episode sentence, alternative solution step sentence) and *conditions* construction is centered on the *main-verb* and its *subject* and *objects*.

Fig. 5 depicts an abstract conceptual model that supports the proposed scenarios language.

3.2. Scenarios relationships

Scenarios might be related to other scenarios, i.e., none of them is entirely independent of the rest of the scenarios (Lee et al., 1998; do Prado Leite et al., 2000). The behavior of a scenario is described through a set of pre-conditions, post-conditions, episodes, alternatives or exceptions, which could be described by simple conditions/states/actions or *references* to other scenarios; the latter case occurs when the condition/state/action is more complex (Cockburn, 2000; Jacobson et al., 1992) or is common to several scenarios (do Prado Leite et al., 2000). The scenario language supports the refinement of complex scenarios in various scenarios according to the following relationships: (1) ***Pre-condition***, a pre-condition is expressed in another scenario, then, this scenario must be executed first; (2) ***Post-condition***, a post-condition is expressed in another scenario, then, this scenario must be executed last; (3) ***Sub-scenario***, an episode is detailed in another scenario, then, this scenario must be executed when the episode is activated; (4) ***Alternate***, an alternate flow of events is detailed in another scenario, then, this scenario must be executed when the alternate path is activated; and (5) ***Exception***, the solution that handles an exception is detailed in another scenario, then, this scenario must be executed last.

In the "*Submit Order*" scenario (Fig. 1a), episodes 10 (LOCAL SUPPLIER BID FOR ORDER), 11 (INTERNATIONAL SUPPLIER BID FOR ORDER) and 12 (PROCESS BIDS), and exception solution 1e1 (REGISTER CUSTOMER) are detailed in other scenarios. Thus, it is possible to identify the sequentially related scenarios.

When facing large systems, the number of scenarios could be unmanageable, and the requirements engineers become sunk in details, losing the global vision of the system. To face this problem, the scenario language supports the construction of *integration scenarios* (do Prado Leite et al., 2000) based on the existing and related scenarios.

These features provide *modularity* through the interconnectivity among related scenarios. For instance, an exception scenario may treat exceptions occurring in different scenarios, a sub-scenario could be included in one or more scenarios, and an integration scenario can give an overview of the relationship among several scenarios.

3.3. Scenarios quality

Firesmith (2007), Pohl and Rupp (2015) and Wiegers and Beatty (2013) discussed several different quality properties that individual requirements and SRSs should exhibit and, concluded that the most important desirable properties are: *unambiguity*, *completeness*, *consistency* and *correctness*. Other desirable properties are *testability*, *traceability*, and *modifiability*. Research conducted by Saavedra et al. (2013) and Zowghi and Gervasi (2003) discussed the impact between quality properties and, suggested that: (1) correctness is a combination of unambiguity, completeness, and consistency; (2) completeness and consistency are decomposed – using *AND* links – in internal and external properties. Evaluating external completeness and external consistency is a hard problem because it depends on external specifications, domain models, and user's satisfaction; consequently, violations are difficult to detect or only with much effort. Zowghi and Gervasi (2003) also argue as increasing the completeness can decrease its consistency and hence affect the correctness. Conversely, improving the consistency can reduce the completeness, thereby again diminishing the correctness of an SRS.

In our work, it is assumed that: (1) *correctness* is the most important quality property of scenarios, and it is *satisficed* (sufficient satisfaction) (Chung et al., 2000) by contributions of *unambiguity*, *completeness* and *consistency* properties; (2) there is a causal relationship between unambiguity, completeness, consistency and correctness; (3) the achievement of these properties demands the fulfillment of other properties; and (4) these properties can be assessed by verification and validation heuristics. Fig. 6 illustrates these quality properties as non-functional requirements (NFRs), and their contributions through a *Soft-goal Interdependency Graph* (*SIG*) (Chung et al., 2000).

3.3.1. Unambiguity

An SRS is *unambiguous* if and only if, every requirement stated therein has only one interpretation (IEEE Computer Society, 1998). Research (Arora et al., 2015; Berry et al., 2012; Femmer et al., 2017; Lami et al., 2004; Wilson et al., 1997) about requirements unambiguity suggested that unambiguity is negatively impacted (HURT — as in Fig. 6) by *vagueness*, *subjectiveness*, *optionality*, *weakness*, *multiplicity*, *implicitly*, and *quantifiability*. Also, it is positively impacted (HELP — as in Fig. 6) by *readability* and *minimality*.

Unambiguity is very difficult to achieve and evaluate because most of its indicators are subjective; however, to detect certain indicators

of violation within requirements statements, some approaches based on *dictionaries* that contain frequently used words or phrases (Femmer et al., 2017; Lami et al., 2004; Wilson et al., 1997), NLP *POS tagging* (Rosadini et al., 2017), and *shallow/dependency parsing* (Rosadini et al., 2017; Femmer et al., 2017), were proposed. These approaches can warn about weak phrases, subjective words, multiple subjects, implicit subjects/objects, and multiple action-verbs.

3.3.2. Completeness

An SRS is *complete* if all relevant requirements are present and each requirement is fully developed (Boehm, 1979). It must not include situations that will not be encountered or capability features that are unnecessary (Wilson et al., 1997).

Internal Completeness is positively impacted (HELP — as in Fig. 6) by individual scenario properties like *atomicity*, *simplicity*, *uniformity*, *usefulness*, and *conceptually soundness*; and by properties that involve a set of related scenarios such as *integrity*, *coherency*, *uniqueness*, and *feasibility*.

To detect certain indicators of violation within individual scenarios and a set of related scenarios, some approaches based on *regular expressions* (Rosadini et al., 2017), *string searching/similarity* (Wilson et al., 1997), and *dependency parsing* (Ciemniewska and Jurkiewicz, 2007), were proposed. These approaches can warn against non-unique scenarios, non-existing scenarios, non-compliance with writing recommendations, and the lack of significant information.

3.3.3. Consistency

An SRS is *consistent* when two or more requirements are not in conflict with one another or with governing specifications and objectives (Boehm, 1979). Inconsistency occurs when two or more users have conflicting requirements, or when one or more requirements override others. Proposals for use case formalization (Cheung et al., 2006; Chu et al., 2017; Denger et al., 2005; Lee et al., 1998; Sarmiento et al., 2015; Sinning et al., 2009; Somé, 2010) through formal representations (e.g., Labeled Transition Systems (Keller, 1976), Statecharts (Harel, 1987) or Petri-Nets (Murata, 1989)) suggested that Internal Consistency is positively impacted (HELP — as in Fig. 6) by *persistence*, *boundedness*, *reversibility* and *liveness*.

Although, consistency is very difficult to evaluate; there are indicators of violation that can be detected by first *mapping* scenarios into *executable models* (e.g., Petri-net or Statecharts) and simulating the behavior of these models using available tools. *Reachability analysis* (Lee et al., 1998) and simulation (Denger et al., 2005) based strategies can warn non-deterministic situations, never enabled operations, and deadlocks.

3.3.4. Correctness

An SRS is *correct* if and only if, every requirement stated therein is one that the software shall meet (IEEE Computer Society, 1998). Incorrectness may occur when the acquired requirements do not accurately reflect the facts or wrongly predict future states. Correctness is difficult to evaluate and achieve; therefore, having more unambiguous, complete, and consistent scenarios contributes (HELP — as in Fig. 6) to more correct scenarios.

3.4. Verification heuristics

Table 2 shows part of the desired quality *properties* of scenarios and their *verification heuristics*. These heuristics are implemented through different techniques. If the result after performing a heuristic is opposite to an expected result, a defect (an indicator of violation) must be reported to the requirements engineer. These defects are categorized as information, warning, or mistake. *Information* reveals that the requirements engineer may have forgotten to specify information related to a scenario element. *Warning* reveals that the requirements engineer may have introduced confusing information or forgot to inform an important scenario element. *Mistake* reveals that the requirements engineer may have introduced incorrect information related to a scenario element. The presence of defects is a strong sign, although not conclusive of incorrectness that must be fixed.

Integrity, *coherency* and *uniqueness* are evaluated by checking a main scenario against the other scenarios. These heuristics were implemented in the C&L.[2]

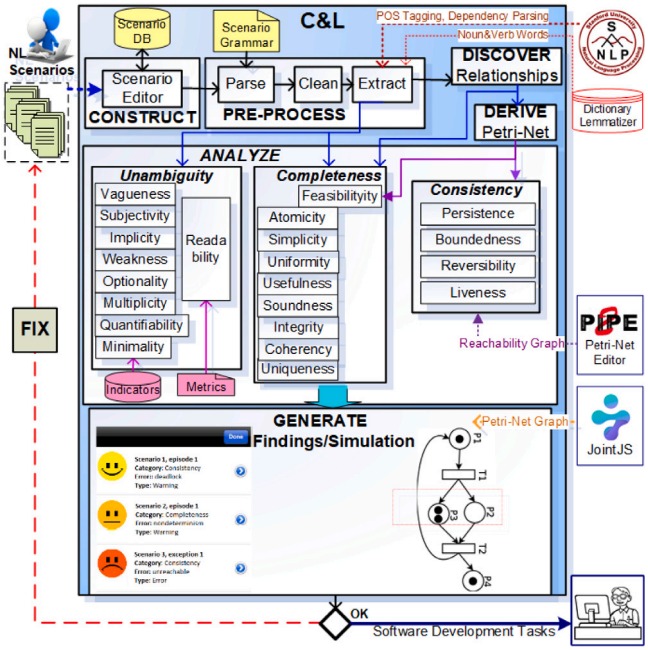


Fig. 7. Overall analysis approach with the C&L tool.

4. Analysis approach

Requirements engineers CONSTRUCT textual scenarios, so our approach PRE-PROCESSES the given scenarios, DISCOVERS their relationships, DERIVES Petri-nets, ANALYZES the scenarios and their equivalent Petri-nets, and GENERATES a report of defects (findings) and a visual representation of the scenarios for simulation. Finally, requirements engineers FIX the defective scenarios. Fig. 7 shows the high-level information flow through our approach, which is hidden for the user and accomplished by the C&L.

4.1. Construct

Requirements engineers start to describe requirements as scenarios. To reduce the superfluous information and maintain consistency between the *vocabulary* of the application and the scenarios, we suggest to follow the scenario construction process proposed by do Prado Leite et al. (2000), i.e., first describe the relevant words or phrases of the application as lexicon symbols (Language Extended Lexicon – LEL), then, derive and evolve the scenarios from the LEL.

4.2. Pre-process

The scenarios are annotated with information.

[2] C&L is available at https://bit.ly/2RqKYId.

Table 2
The quality properties of scenarios and their verification heuristics: unambiguity, completeness and consistency.

Property	Description	Verification heuristic
Vagueness	The sentence contains words or phrases having a non-uniquely quantifiable meaning (Lami et al., 2004).	Check that a sentence does not contain vague terms (e.g., adaptability, additionally).
Subjectiveness	The sentence contains words or phrases expressing personal opinions or feelings (Lami et al., 2004).	Check that a sentence does not contain comparative/superlative adverbs/adjectives (e.g., similar, better).
Optionality	The sentence contains words that give the developer latitude in satisfying the specification statements that contain them (Wilson et al., 1997).	Check that a sentence does not contain optional words (e.g., as desired, at last).
Weakness	The sentence contains clauses that are apt to cause uncertainty and leave room for multiple interpretations (Wilson et al., 1997).	Check that a sentence does not contain weak terms (e.g., can, preferred).
Multiplicity	The sentence has more than one main verb or subject (Adapted from Lami et al., 2004).	Check that a sentence does not contain conjunction or disjunction of verbs or subjects (e.g., and, or, and/or).
Implicitly	The sentence does not specify the subject or object by means of its specific name but uses pronoun or indirect Lami et al. (2004) (Wilson et al., 1997).	Check that a sentence does not contain implicit words (e.g., anyone, he, her);
Quantifiability	Terms used for quantification can lead to ambiguity if not used properly (Arora et al., 2015).	Check that a sentence uses quantification words in a clear way (e.g., all, any, few).
Minimality	A sentence contains nothing more than basic attributes (Lucassen et al., 2015).	Check that a sentence does not contain additional information (Text after a dot, hyphen, semicolon, or other punctuation mark)
Atomicity	A scenario expresses exactly one situation (Adapted from Lucassen et al., 2015).	Check that the Title defines exactly one situation (do Prado Leite et al., 2000). Check that the Goal satisfies exactly one purpose (do Prado Leite et al., 2000). Check that the Title contains a verb in infinitive form and an object (Cockburn, 2000) (do Prado Leite et al., 2000).
Simplicity	A scenario should be as readable as possible.	Check that an Episode-Sentence is described from user's point of view (Subject + present simple tense and active form of verb + Object) or in another scenario (infinitive verb – base form + Object) (Ciemniewska and Jurkiewicz, 2007) (Cockburn, 2000) (do Prado Leite et al., 2000) (Yue et al., 2013) Check that an Alternative-Solution-Step-Sentence is described from user's point of view (present simple tense and active form of verb + Object) or in another scenario (infinitive base form of verb + Object). Optionally, it contains a Subject (Ciemniewska and Jurkiewicz, 2007) (Cockburn, 2000) (do Prado Leite et al., 2000) (Yue et al., 2013). Check that the Title does not contain extra unnecessary information (Phalp et al., 2007). Check that an Episode coincidence only takes place in different situations (do Prado Leite et al., 2000). Check that episodes involving validation are described using the verbs verify/validate/ ensure/establish and followed by that; i.e., avoid verbs like check/see followed by If/Whether. Complicated validation steps can confuse the user and be difficult to understand Ciemniewska and Jurkiewicz (2007) (Cockburn, 2000) (do Prado Leite et al., 2000). Check that a nested IF statement is not used in a Conditional Episode, i.e., it can confuse the user and be difficult to read (Ciemniewska and Jurkiewicz, 2007) (Cox et al., 2003). Check that an alternative is handled by a simple action (do Prado Leite et al., 2000), i.e, if the interruption is treated by a sequence of steps (> 3), this sequence should be extracted to a separate scenario (Ciemniewska and Jurkiewicz, 2007). Check that every alternative flow returns to a specific episode of the main flow or finishes the scenario (Van Galen, 2012).

(*continued on next page*)

Table 2 (*continued*).

Property	Description	Verification heuristic
Uniformity	Each scenario element should be described with significant information.	Ensure that the Title is present (do Prado Leite et al., 2000). Ensure that the Goal is present (do Prado Leite et al., 2000). Check the existence of more than one Actor per Scenario (do Prado Leite et al., 2000). Ensure that the Context contains its relevant sub-components (do Prado Leite et al., 2000). Check the existence of more than one Episode per Scenario (do Prado Leite et al., 2000). Ensure that an Episode contains its relevant parts (do Prado Leite et al., 2000). Ensure that a non-sequential episodes construct has a begin and end keywords (e.g., #). Ensure that an Alternative contains its relevant parts (do Prado Leite et al., 2000).
Usefulness	A scenario does not contain superfluous information, i.e., there should be consistency among scenario components. (Adapted from Anda et al., 2009).	Check that every Actor participates in at least one episode (do Prado Leite et al., 2000). Check that every Resource is used in at least one episode (do Prado Leite et al., 2000). Check that every Actor mentioned in episodes is included in the Actor section (do Prado Leite et al., 2000) or is the System (Yue et al., 2013) or, it is included in Resources. Check that every Resource mentioned in episodes is included in the Resource (do Prado Leite et al., 2000) section or, it is included in Actors. Check that every Actor mentioned in alternatives is included in the Actor section (do Prado Leite et al., 2000) or is the System (Yue et al., 2013) or, it is included in Resources. Ensure that step numbering between the main flow and alternative flows is consistent (Liu et al., 2014). Check the existence of more than 2 and less than 10 episodes per scenario (Ciemniewska and Jurkiewicz, 2007) (Cockburn, 2000) (do Prado Leite et al., 2005).
Conceptually soundness	Internal scenario elements are semantically coherent, i.e., elements satisfy the scenario goal (do Prado Leite et al., 2000).	Check that the Title describes the Goal. Ensure that the Episodes contain only actions to be performed (do Prado Leite et al., 2000). Ensure that the Alternatives contain only actions to be performed (do Prado Leite et al., 2000).
Integrity	Whenever a scenario references to another scenario, the related scenario should exist within the set of scenarios.	Check that every included scenario (Pre-condition, Post-condition, Episode sentence, Alternative solution) exists within the set of scenarios (do Prado Leite et al., 2000). Ensure that actions present in the Pre-conditions are already performed (do Prado Leite et al., 2000). Check that Episode coincidence only takes place in different scenarios (do Prado Leite et al., 2000).
Coherency	Internal components of explicitly related scenarios should be precise and use a common terminology, e.g., pre-conditions of related scenarios are coherent.	Check coherence between Pre-conditions in related scenarios (do Prado Leite et al., 2000). Check that the Geographical and Temporal locations of the related scenarios are equal or more restricted than those of the main scenario (do Prado Leite et al., 2000). Check that referenced scenarios do not reference the main scenario (Sarmiento et al., 2015) (adapted from Liu et al., 2014).
Uniqueness	A scenario is unique when no other scenario is the same or too similar, i.e., duplicates are avoided because they are a source of inconsistencies (Adapted from Lucassen et al., 2015).	Check that the Title of a scenario is not already included in another scenario. Check that the Goal of a scenario is not already included in another scenario. Check that the Pre-condition of a scenario is not already included in another scenario. Check that the set of Episodes of a scenario is not already included in another scenario. Check that two scenarios do not have similar Titles.
Feasibility	It is possible to perform each operation described in a scenario.	Check that is possible to derive an initial system design from related scenarios (Denger et al., 2005). Check that the initial system design does not contain isolated sub-systems (Lee et al., 2001).
Persistence	For any two enabled operations, the execution of one operation will not disable the other.	Check the absence of non-deterministic execution paths, i.e., a set of episodes are simultaneously enabled by common pre-conditions, and only one of them may happen (Lee et al., 1998).
Boundedness	This property refers to the limited capacity of a communication channel or resource.	Check the absence of overflow, i.e., the number of elements in some communication channel or resource exceeds a finite capacity (Lee et al., 1998).
Liveness	Every operation can be executed in the future	Check the absence of paths to *deadlocks* (Lee et al., 1998), e.g., it may occur when an alternative flow does not return to the main flow, does not finish the scenario or is not treated/handled. Check the absence of never enabled operations, e.g., when the pre-conditions of an operation are never fulfilled.
Reversibility	This property guarantees that the described behavior reaches its initial state again	Check that is possible to return to the initial state from any path.

4.2.1. Parse

This procedure reads a textual scenario and parses it into a structured scenario (conform to the model of Fig. 5). After a detailed analysis of nine different variants of scenario/use-case templates (Jacobson et al., 1992; Cockburn, 2000; do Prado Leite et al., 2000; Insfrán et al., 2002; Kruchten, 2003; Somé, 2010; Sinha et al., 2010; Yue et al., 2013; Chu et al., 2017), we define a *grammar* (Fig. 8a) that accepts textual scenarios described conform to the common sections of the general template described in Fig. 2. This grammar allows us to: (1) identify the main sections of a scenario and (2) parse the sections into their relevant components and sub-components. Fig. 8b shows the information to be found within scenario statements using NLP-based strategies.

For instance, this pattern "<step> (<delimiter> | <white-space>) <episode-sentence>" parses a textual episode into a structured episode composed of a *step* and a *sentence*.

4.2.2. Clean

To improve the efficacy of scenario transformation procedures and the accuracy of NLP analysis tools, it is necessary to clean the scenario statements, i.e., to lowercase every character and to remove irrelevant information such as empty lines, parenthesized ([…], (…), {…})

(a)

```
<Scenario> = "TITLE:" <Title> + "GOAL:" <Goal> + "CONTEXT:" <Context> + [ "RESOURCES:" <Resource>] + "ACTORS:" <Actor> + "EPISODES:" <Episodes> +
             "ALTERNATIVES/EXCEPTIONS:" {<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> + {<punctuation-connective> <item>}
<Actor> = <item> + {<punctuation-connective> <item>}
<Episodes> = <Group> | <Episodes> <Group>
  <Group> = <Sequential-Group> | <Non-Sequential-Group>
  <Sequential-Group> = <Episode> <Episode> | <Sequential-Group> <Episode>
  <Non-Sequential-Group> = {<Episode>} <non-sequential-struct-begin> <Episode-Series> <non-sequential-struct-end> {<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>} + <CR>
   <Simple-Episode> = <step> + (<delimiter> | <white-space>) + <episode-sentence>
   <Conditional-Episode> = <step> + (<delimiter> | <white-space>) + (((("IF" | "WHEN") + <Condition> + ("THEN" | ",") + <episode-sentence>) |
                                                                     (<episode-sentence> + ["," + ("," + "IF" | "WHEN") + <Condition>))
   <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> = (<Alternative-IF-THEN> | <Alternative-Multi-Line>) + ["POST-CONDITION:" <Post-condition>]
  <Alternative-IF-THEN> = <step> + <ref> + (<delimiter>|<white-space>) + "IF" + <Cause> + "THEN" + [":"] + [<CR>] +
                          <Solution>
  <Alternative-Multi-Line> = <step> + <ref> + (<delimiter>|<white-space>) + <Cause> + [":"] + [<CR>] +
                             <Solution>
  <step> = <digit>{<digit>}
  <ref> = (<delimiter> + <digit>{<digit>} | [<delimiter>] + <alphabetic_character>)
  <Condition> = <atomic-sentence> | <Condition> <connective> <Condition>
  <Cause> = <atomic-sentence> | <expression> | <Cause> <connective> <Cause>
  <Solution> = <Solution-Step> + {<CR> <Solution-Step>}
  <Solution-Step> = [(<step> | <step> + <ref> + <ref>) + (<delimiter> | <white-space>)] + <solution-sentence>
  <connective> = "AND" | "OR"
  <punctuation-connective> = "," | ";"
  <non-sequential-struct-begin> = "#"
  <non-sequential-struct-end> = "#"
  <delimiter> = "." | ":"
  <item> = <alphabetic_character> {<all_characters>}
  <digit> = ? all digit characters ?
  <alphabetic_character> = ? all alphabetic characters ?
  <all_characters> = ? all visible characters ?
  <CR> = ? carriage return ?
```

(b)

```
<Title> = ([<Subject>] + action-verb + object) | Phrase
<Goal> = [<Subject>] + verb + predicate | phrase
<state> = (<Subject> + state-verb + predicate) | phrase
<atomic-sentence> = (<Subject> + linking-verb + predicate) | phrase
<episode-sentence> = (([<Subject>] + action-verb + (object + [prepositional-phrase] | prepositional-phrase)) | <Title>) + {<Constraint>}
<solution-sentence> = ([<Subject>] + action-verb + (object + [prepositional-phrase] | prepositional-phrase)) | <Title> | (( "GOTO"|"GO" | "BACK" | "RETURN"
                    | "RESUME") + ["TO"] + ["STEP"|"EPISODE"] + <Step>) | (("SYSTEM"|"USE CASE"|"SCENARIO") + ("END"["S"]| "TERMINATE"["S"]|"FINISH"["ES"]))
<Constraint> = ([<Subject>] + ["MUST"] + ["NOT"] + predicate) | phrase
<Subject> (<Actor> | <Resource> | "System")
```

Fig. 8. A grammar for scenario/use-case descriptions (a) and the information to be found in scenario/use-case statements (b).

comments, URLs, HTML tags, punctuation, and bullets. We use regular expression matching to perform the cleaning tasks.

For instance, this pattern "\ (. * \)" searches for extra unnecessary information described between parentheses in a scenario statement (e.g., title or goal).

4.2.3. Extract

This procedure (as shown in Fig. 9) *tokenizes* each scenario statement into *tokens* and its corresponding *words*. Next, for each token, it generates its base/root form – *lemma* and POS tag. It acts as a basis for the next step – *dependency parsing*. This step gets the grammatical relations – *typed dependencies* (de Marneffe et al., 2014) between individual words in a sentence. Finally, from the *typed dependencies*, it extracts relevant information such as *subjects*, *objects*, and *action-verbs*.

Due to the imprecision of the *POS tagging* phase, we improved the accuracy of the parsing phase by adding a fourth step with simple rules to adjust *words* to their correct POS tags. This strategy checks that a *word* in a sentence is correctly tagged as a *noun* (NN, NNS), *verb* (VB, VBP, VBZ), *preposition* (IN), or *adjective* (JJ), by verifying that the *word* ∈ *NOUN_AND_VERB* dictionary (words that are both *noun* and *verb*)[3] and looking at the POS tags of *neighboring* words.

Fig. 10a illustrates a few rules for adjusting POS tags (the rule set is available in Appendix A).[4] For example, "*types*" and "*presses*" can be tagged as nouns (NN) as in *"User types in the numbers of his PIN and presses the Enter button"*. Then, an *adjusting rule* must put the correct tags (verb/VBZ), because "*types*" and "*presses*" ∈ *NOUN_AND_VERB*, and they are between a noun(NN) and a preposition(IN), and a conjunction(CC) and a determiner(DT), respectively.

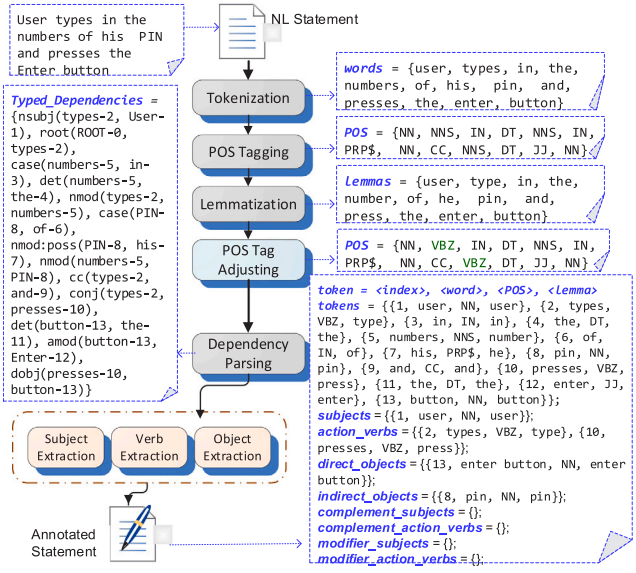
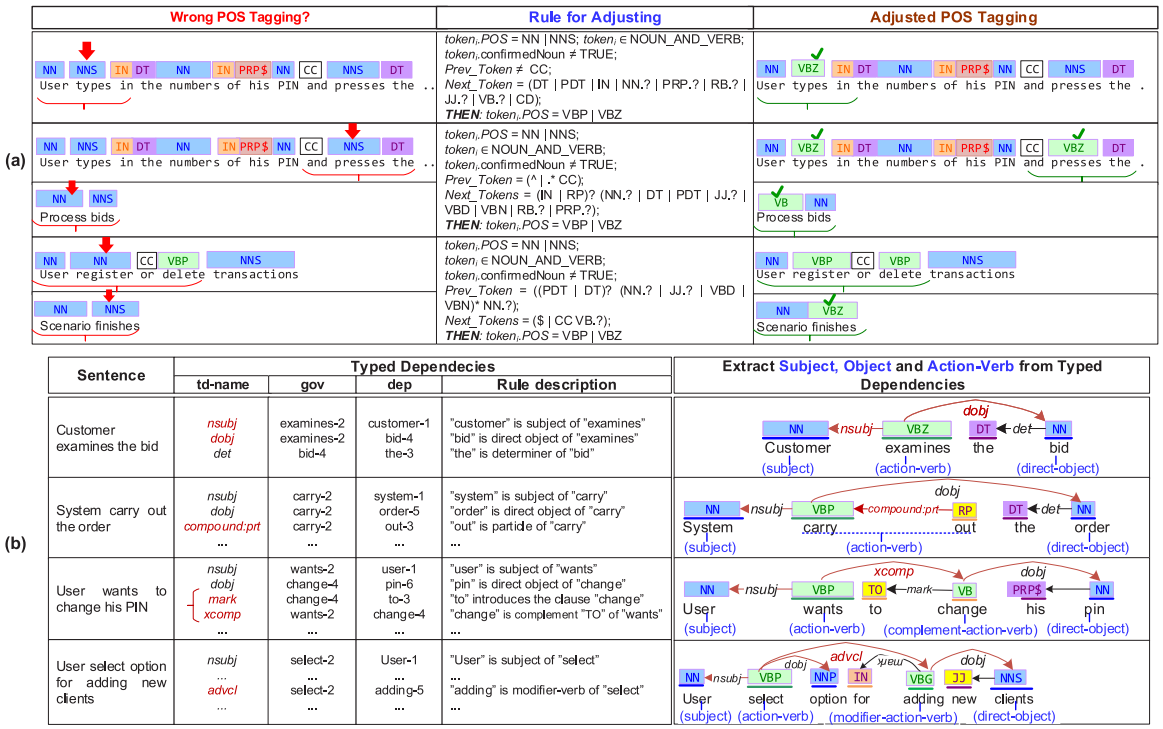


Fig. 9. NLP pipeline for information extraction.

The last step analyzes the *typed dependencies* to highlight nouns and verbs functioning as subjects, objects, and action-verbs within a sentence. This step verifies: (1) *nsubj* (nominal subject), *nsubjpass*

[3] Dictionary is available at https://bit.ly/3vB2719.

[4] Appendix A is available at https://bit.ly/2ZIcr8O.



Sentence	Typed Dependencies			
	td-name	gov	dep	Rule description
Customer examines the bid	*nsubj* *dobj* *det*	examines-2 examines-2 bid-4	customer-1 bid-4 the-3	"customer" is subject of "examines" "bid" is direct object of "examines" "the" is determiner of "bid"
System carry out the order	*nsubj* *dobj* *compound:prt* ...	carry-2 carry-2 carry-2 ...	system-1 order-5 out-3 ...	"system" is subject of "carry" "order" is direct object of "carry" "out" is particle of "carry" ...
User wants to change his PIN	*nsubj* *dobj* *mark* *xcomp* ...	wants-2 change-4 change-4 wants-2 ...	user-1 pin-6 to-3 change-4 ...	"user" is subject of "wants" "pin" is direct object of "change" "to" introduces the clause "change" "change" is complement "TO" of "wants" ...
User select option for adding new clients	*nsubj* ... *advcl* ...	select-2 ... select-2 ...	User-1 ... adding-5 ...	"User" is subject of "select" ... "adding" is modifier-verb of "select" ...

Fig. 10. Adjusting noun and verb POS tags (a) and Extracting information from dependency parsing (b).

(passive nominal subject) and *dep* (dependent) relations to highlight subjects and action-verbs; (2) *dobj* (direct object) and *dep* (dependent) relations to highlight direct-objects and action-verbs; (3) *iobj* (indirect object), *pobj* (object after a preposition) and *nmod* (nominal modifier) relations to highlight indirect-objects and action-verbs; (4) *ROOT* relation to highlight verb phrases functioning as action-verbs; (5) *conj* (conjunction) and *nmod* (nominal or oblique – *obl* modifier) relations to highlight subjects, objects and action-verbs; (6) *compound* (compound modifier), *poss* (possession modifier) and *nummod* (numerical modifier) relations to highlight compound nouns functioning as subjects or objects (e.g., broker system); (7) *compound:prt* (phrasal verb particle) relation to highlight compound verbs functioning as action-verbs (e.g., carry out); (8) *xcomp* (open clausal complement) relations to highlight verbs without its own subject, and functioning as complement-action-verbs (subordinate); (9) *advcl* (adverbial clause modifier) relation to highlight verbs that modify a verb or other predicate, and functioning as modifier-action-verbs; (10) *acl* (clausal modifier of a noun) and *dep* (dependent) relation to highlight verbs that modify a noun, and functioning as modifier-action-verbs (coordinate); and (11) *obj* (object) relation to highlight direct-objects or indirect-objects.

Fig. 10b illustrates a few rules for extracting subjects, action-verbs, and objects (direct and indirect) from typed dependencies (the rule set is available in Appendix B).[5]

4.3. Discover

Although such subsets of scenarios might seemingly be independent, they are rarely truly independent in practice (Lee et al., 1998) (do Prado Leite et al., 2000). Thus, these scenarios might interact with other scenarios by *explicit* or *non-explicit* relationships. *Procedure* 1 reads any two scenarios, compares their statements, and identifies the type of relationship among them. The following subsections detail its steps.

Procedure 1 Identify the Type of Relationship among any Two Scenarios

Input: Scenarios S_i and S_j
Output: $type\_of\_relationship$, $\{scenario\_component\}$
//EXPLICIT RELATIONSHIPS
for each condition $c \in S_i.context.pre\_condition$ **do**
  if c contains $S_j.title$ **then**
    Return {"PRE-CONDITION", $\{c\}$}
for each condition $c \in S_i.context.post\_condition$ **do**
  if c contains $S_j.title$ **then**
    Return {"POST-CONDITION", $\{c\}$}
for each episode $e \in S_i.episodes$ **do**
  if $e.sentence$ contains $S_j.title$ **then**
    if e is inside a non-sequential group **then**
      Return {"NON-SEQUENTIAL", $\{e\}$}
    else
      Return {"SUB-SCENARIO", $\{e\}$}
for each alternative $a \in S_i.alternatives$ **do**
  for each solution_step $s \in a.solution$ **do**
    if $s.solution\_sentence$ contains $S_j.title$ **then**
      Return {"ALTERNATIVE", $\{a\}$}
//NON-EXPLICIT RELATIONSHIPS
Calculate $I_{i,j} \leftarrow$ Proximity-Index(S_i, S_j)
if $I_{i,j} \geqslant 0.5$ **then**
  $Set\text{-}Pre\text{-}Post \leftarrow (S_i.context.pre\_condition \bigcup S_i.episodes.pre\_condition) \bigcap (S_j.context.pre\_condition \bigcup S_j.episodes.pre\_condition)$
  if |Set-Pre-Post| > 0 **then**
    Return {"NON -DETERMINISM", $\{Set\text{-}Pre\text{-}Post\}$}
  $Set\text{-}Pre\text{-}Post \leftarrow ((S_i.context.pre\_condition \bigcup S_i.episodes.pre\_condition) \bigcap (S_j.context.post\_condition \bigcup S_j.episodes.post\_condition)) \bigcap ((S_i.context.post\_condition \bigcup S_i.episodes.post\_condition) \bigcap (S_j.context.pre\_condition \bigcup S_j.episodes.pre\_condition))$
  if |Set-Pre-Post| > 0 **then**
    Return {"SYNCHRONIZATION", $\{Set\text{-}Pre\text{-}Post\}$}
Return {"", {}}

[5] Appendix B is available at https://bit.ly/3bxGRjQ.

4.3.1. Explicit relationships

When a scenario S_i includes the title (often in capital letters) of another scenario S_j in its description, then, S_i and S_j might interact by:

Sequential Relationships: S_i includes the title of S_j in a pre-condition, post-condition, episode, alternate, or exception (Section 3.2), then, S_i and S_j will be executed in sequential order – precedence order.

Non-sequential relationships: S_i includes the title of S_j in one of its episodes e, and $e \in \{\#e_1, e_2, \ldots, e_n\#\}$ a non-sequential group of episodes, then, S_j will be activated from S_i, and executed in parallel or concurrently with other episodes inside $\{\#e_1, e_2, \ldots, e_n\#\}$.

In Fig. 1a, the scenarios PROCESS BIDS and REGISTER CUSTOMER are sequentially related to the "*Submit Order*" scenario; and the scenarios LOCAL SUPPLIER BID FOR ORDER and INTERNATIONAL SUPPLIER BID FOR ORDER are non-sequentially related to the "*Submit Order*" scenario. However, the given scenarios might interact non-explicitly.

4.3.2. Non-explicit relationships

Two or more scenarios might interact non-explicitly and in a non-sequential order, which can lead to *non-deterministic situations* or *deadlocks*.

Any two scenarios are likely non-explicitly and non-sequentially related when they share common portions in their descriptions – *Proximity Index* (do Prado Leite et al., 2005), 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).

Let I_{ij} (1) be the Proximity Index between any two scenarios S_i and S_j, where α, β and μ are weight factors.

$$
\begin{aligned}
I_{i,j} &= \frac{\alpha^* A\cap_{i,j} + \beta^* R\cap_{i,j} + \mu^* C\cap_{i,j}}{\alpha^* A\cup_{i,j} + \beta^* R\cup_{i,j} + \mu^* C\cup_{i,j}} \\
A\cap_{i,j} &= |Actors(S_i) \cap Actors(S_j)| \\
A\cup_{i,j} &= |Actors(S_i) \cup Actors(S_j)| \\
R\cap_{i,j} &= |Resources(S_i) \cap Resources(S_j)| \\
R\cup_{i,j} &= |Resources(S_i) \cup Resourcse(S_j)| \\
C\cap_{i,j} &= |Context(S_i) \cap Context(S_j)| \\
C\cup_{i,j} &= |Context(S_i) \cup Context(S_j)|
\end{aligned}
\tag{1}
$$

If the *proximity index* $I_{i,j} \geq 0.5$, there is an indication that S_i and S_j must be analyzed more deeply, i.e., their pre-conditions and post-conditions must be compared. These scenarios might interact non-explicitly by:

Non-determinism: If two scenarios S_i and S_j share a subset of pre-conditions, then, S_i and S_j might interact concurrently.

Synchronization: If a scenario S_i includes among its post-conditions a pre-condition of another scenario S_j, and S_j includes among its post-conditions a pre-condition of S_i, then, S_i and S_j might interact concurrently.

For instance, the scenarios LOCAL SUPPLIER BID FOR ORDER (S_1) and INTERNATIONAL SUPPLIER BID FOR ORDER (S_2) (as in Fig. 1a) are related by *non-determinism* because they have the same pre-condition ("An order has been broadcasted"), and their degree of proximity $I_{1,2} = 0.5$, i.e., for $\alpha = \beta = \mu = 1 \Rightarrow A\cap_{1,2} = 1$, $A\cup_{1,2} = 3$, $R\cap_{1,2} = 0$, $R\cup_{1,2} = 0$, $C\cap_{1,2} = 1$, $C\cup_{1,2} = 1$, and $I_{1,2} = 0.5$.

4.4. Derive

Once a textual scenario is pre-processed and its relationships identified, it is possible to automatically generate a Petri-net representation from them.

4.4.1. Transforming scenarios into Petri-nets

A Petri-net *PN* is derived from a structured scenario S by identifying the *events* and their guard *conditions* and *constraints* in S, i.e., the *event occurrences* in the scenario triggering – *initial event*, episodes, alternative flows, non-sequential constructs, and scenario completion – *final event*. For each *event* occurrence, a *transition* is created. *Input places* are created to denote the locations of its *conditions, pre-conditions, causes, and constraints*. *Output places* are created to denote the locations of its *post-conditions*. *Event* labels, *condition* labels and *constraint* labels are assigned to these *transitions* and *places*, accordingly. The initial marking M_0 of *PN* is then created to denote the scenario's initial state, i.e., tokens are added into input places that represent the conditions, pre-conditions, constraints or causes. Execution of the scenario begins at M_0, which semantically means the availability of the resources and pre-conditions. It ends at the same marking that semantically means the release of these resources and pre-conditions. The transformation procedure is available in Appendix C.[6]

The *first step* of this procedure applies mapping rules to translate the scenario sections and components into Petri-net elements. Fig. 11 visually depicts these rules using a structure composed of a left-hand side and a right-hand side (LHS ⇒ RHS). LHS is the conditional part of the rule (scenario element), and RHS is the expected result of the rule (sub-Petri-net). To preserve the event sequences, we add appropriate input dummy places, output dummy places, or dummy transitions into the sub-Petri-nets. *Dummy transitions* to denote the scenario triggering, scenario completion, condition verification, and fork&join synchronization events. *Dummy places* to link sub-Petri-nets derived from sequential events (e.g., episode 1 and episode (2) and fork&join synchronization events; enabling the information flow.

The *second step* composes the sub-Petri-nets generated from the scenario sections into a whole Petri-net by applying *Linking* and *Fusion* operations.

Definition 4.1 (*Linking Sub-Petri-Nets*)**.** Two sub-Petri-nets SPN_i and SPN_j are connected by fusing the output dummy place of SPN_i with the input dummy place of SPN_j.

Definition 4.2 (*Fusion of Places*)**.** Two places p_i and p_j with identical labels are fused by merging their input and output arcs into p_i and eliminating p_j.

For illustration, Fig. 12a shows the equivalent Petri-net derived from the "*Submit Order*" scenario (Fig. 1a). It consists of 20 *event occurrences* as described in the following: 1 scenario triggering, 12 episodes (1, 2, 3, …, 12), 1 scenario completion, 2 exceptions – 1 action per solution (1e1 and 12e), 2 alternate flows – 1 action per flow (2a1 and 4a1), and 1 non-sequential construct – fork&join for synchronizing the non-sequential episodes 10 and 11. We derive a Petri-net by creating transitions $T0$, $T1$, $T2$, …, $T12$, $T13$, $T1e1$, $T12e$, $T2a1$, $T4a1$, $Fork\_1$ and $Join\_1$ to denote these events; and linking input and output places to them, which denote the pre-conditions, post-conditions, conditions and causes of the events. Event labels are assigned to each transition by concatenating the character "T" and the <Step> or <Step><Ref>[<Ref>] attributes of an episode or alternative solution step, respectively. For instance, transition "$T1$" corresponds to the event in the episode "1. *The Customer loads the login page*". Transitions $T0$ and $T13$ denote the scenario triggering and completion, respectively. Transitions $Fork\_1$ and $Join\_1$ denote the synchronization of the non-sequential events $T10$ and $T11$. Input places denoting scenario pre-conditions, and output places denoting scenario post-conditions, are linked to transitions $T0$ and $T13$, respectively. Input and output places are labeled with the same label of guard conditions and constraints. Additionally, we create the dummy transitions $T7\_WHILE$, $Begin$, and End for dealing with the loop event $T7$; and the dummy transition $T8a\_unhandled$ that denotes the unhandled event raised from the event $T8$.

[6] Appendix C is available at https://bit.ly/2Nwwepi.

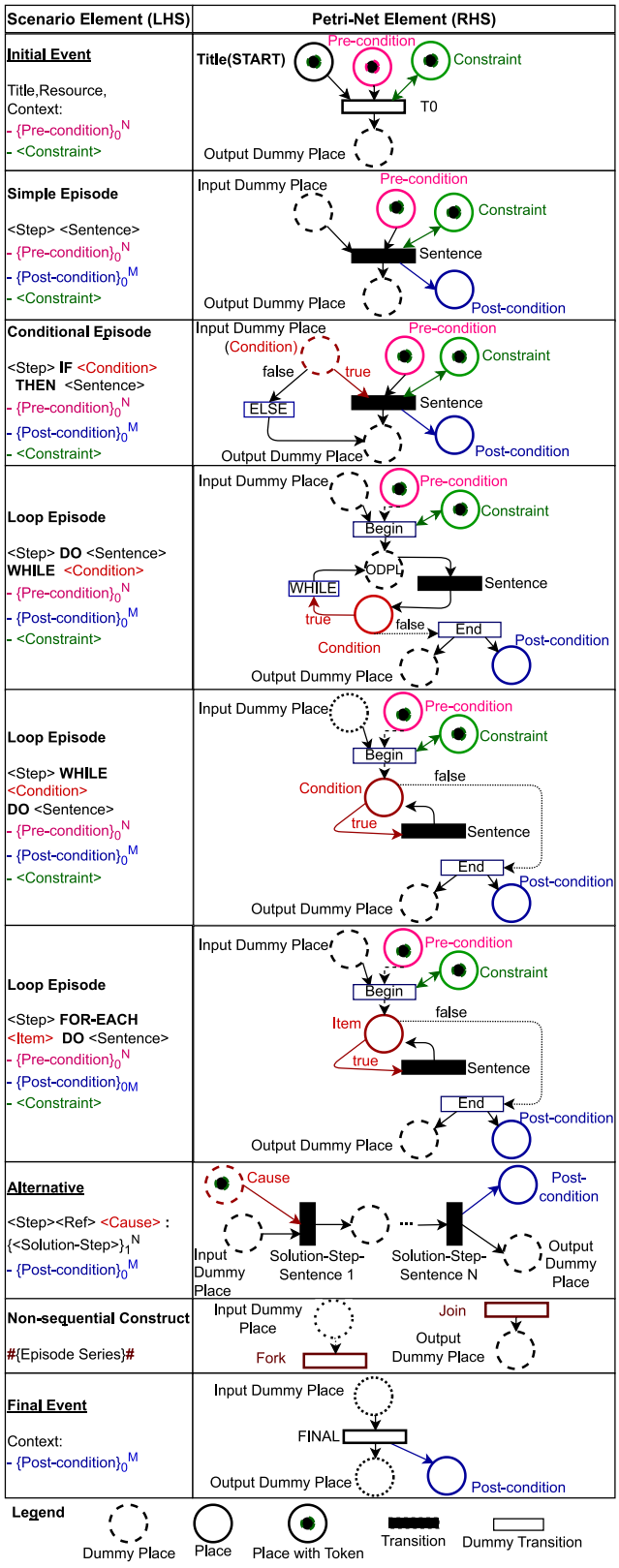


Fig. 11. Mapping scenario elements into Petri-net elements.

4.4.2. Integrating related Petri-nets

When a scenario S is translated into a Petri-net *IPN*, its explicitly and non-explicitly related scenarios are also translated into partial Petri-nets. To simulate the behavior of this set of related scenarios, these partial Petri-nets must be integrated into the consistent whole integrated Petri-net *IPN*. *Procedure* 2 produces an integrated Petri-net from a given scenario S and its related scenarios $\{S_1, S_2, \ldots, S_n\}$.

Procedure 2 Integrate related Petri-nets from a Given Scenario and a Set of Scenarios

Input: Scenario S, Set of scenarios $SS = \{S_1, S_2, \ldots, S_n\}$
Output: Integrated Petri-Net $IPN = \{P, T, F, W, M_0\}$
IPN ← Transform-Scenario-into-Petri-Net(S)
for each scenario $S_i \in SS$ **do**
  Type-Rel, *Scenario-Component* ← Identify-Relationship-Scenarios(S, S_i)
  //EXPLICIT SEQUENTIAL AND NON-SEQUENTIAL RELATIONSHIPS
  if *Type-Rel* = "PRE-CONDITION" | "POST-CONDITION" | "SUB-SCENARIO" | "NON-SEQUENTIAL" | "ALTERNATIVE" **then**
    PN ← Transform-Scenario-into-Petri-Net(S_i)
    Remove input arcs of the first input dummy place (Start) of *PN*
    if *Type-Rel* = "SUB-SCENARIO" | "NON-SEQUENTIAL" | "ALTERNATIVE" **then**
      Remove tokens of the first input dummy place (Start) of *PN*
      Substitute the corresponding Transition (Scenario-Component) of the *IPN* with *PN* (Definition 4.3)
    if *Type-Rel* = "PRE-CONDITION" **then**
      Remove the arc between the last dummy transition (Final) and the corresponding Input Place of *IPN*
      Substitute the corresponding Input Place (Scenario-Component) of the *IPN* with *PN* (Definition 4.4)
      Link the last dummy transition (Final) of *IPN* and the first input dummy place (Start) of *PN*
    if *Type-Rel* = "POST-CONDITION" **then**
      Substitute the corresponding Output Place (Scenario-Component) of the *IPN* with *PN* (Definition 4.5)
  //NON-EXPLICIT AND NON-SEQUENTIAL RELATIONSHIPS
  if *Type-Rel* = "NON-DETERMINISM" | "SYNCHRONIZATION" **then**
    PN ← Transform-Scenario-into-Petri-Net(S_i)
    Integrate *PN* and *IPN* by fusing the common places (generated from Pre-Condition or Post-Condition – Scenario) among them (Definition 4.2)
Return *IPN*

The *first step* of this procedure translates each explicitly related scenario into a Petri-net *PN*. Then, each *PN* must be *substituted* into the corresponding place or transition of *IPN*. This step is the *substitution of places* or *transitions*.

The *second step* translates each non-explicitly related scenario into a Petri-net *PN*. Between *PN* and *IPN*, there are *places* with the same label that denotes the same scenario pre-condition or post-condition, and they need to be uniquely represented. This step is the *fusion of common places*.

Definition 4.3 (*Substitution of Transition*)**.** Any transition t can be replaced by a Petri-net *PN* by fusing the input dummy place of t with the first input place (Start) of *PN* and linking the last dummy transition (Final) of *PN* to the output dummy place of t.

Definition 4.4 (*Substitution of Input Place*)**.** Any input place p can be replaced by a Petri-net *PN* by linking the last dummy transition (Final) of *PN* to p.

Definition 4.5 (*Substitution of Output Place*)**.** Any output place p can be replaced by a Petri-net *PN* by fusing p with the first input dummy place (Start) of *PN*.

Revisiting the "*Submit Order*" scenario S (as in Fig. 1a) and its corresponding Petri-net *IPN* (as in Fig. 12a). The exception solution step 1*e*1 and episodes 10, 11 and 12 in S are detailed in other scenarios

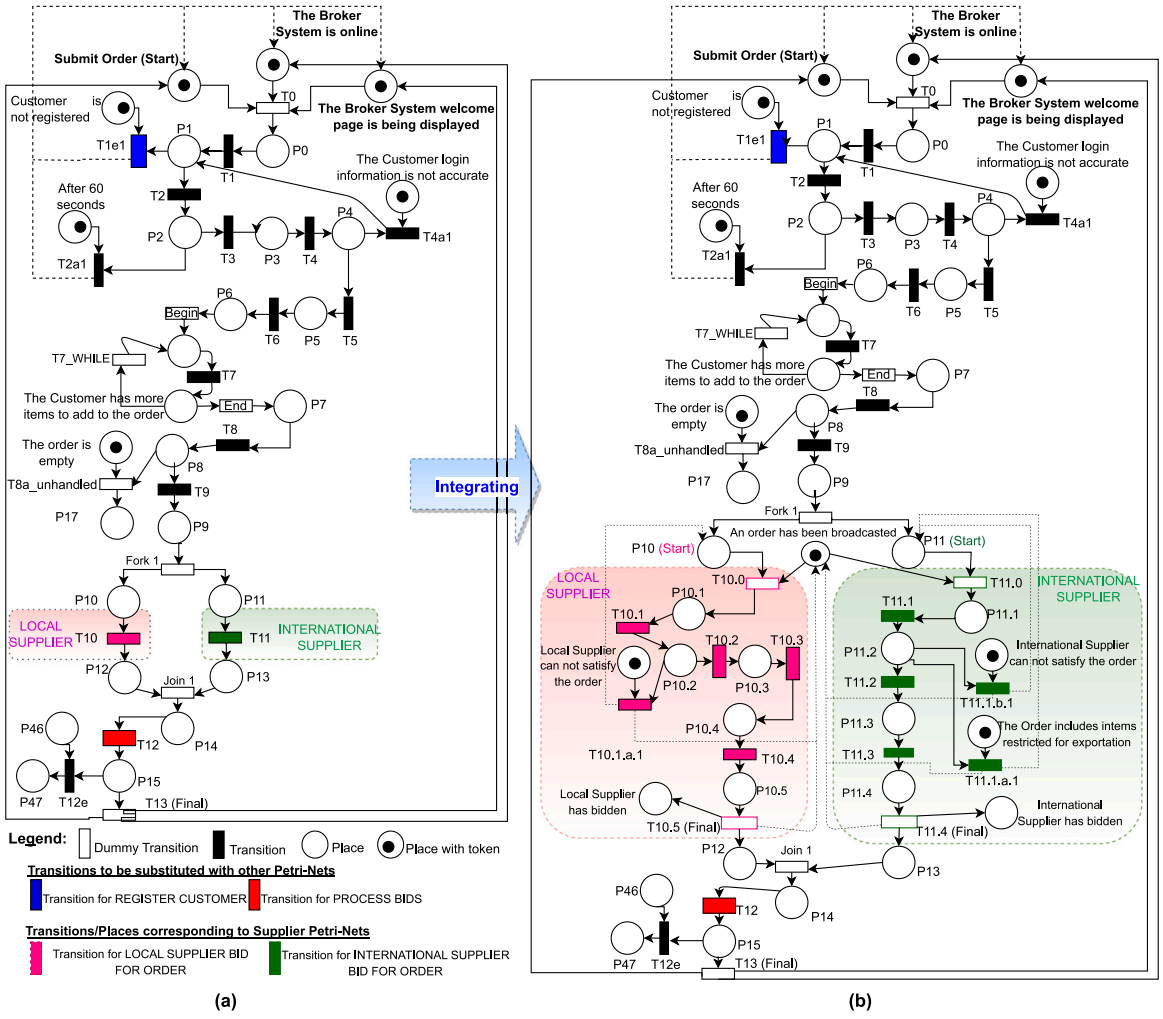


Fig. 12. Mapping the "Submit Order" scenario into a Petri-net, (a) and integrating its related scenarios into a consistent whole Petri-net (b).

S_1 (REGISTER CUSTOMER), S_2 (LOCAL SUPPLIER BID FOR ORDER), S_3 (INTERNATIONAL SUPPLIER BID FOR ORDER) and S_4 (PROCESS BIDS), respectively. When S is translated into *IPN*, the events and their guard conditions in S are mapped into transitions and places in *IPN*, respectively. Thus, Petri-nets should be derived from the related scenarios $\{S_1, S_2, S_3, S_4\}$ and substituted into the corresponding transitions $T1e1$, $T10$, $T11$ and $T12$ of *IPN*. Fig. 12a highlights these transitions and Fig. 12b depicts how transitions $T10$ and $T11$ are substituted by the Petri-nets derived from S_2 and S_3, respectively.

4.4.3. Preservation of properties

The *transformation procedure* preserves the scenario event sequences and conditions into the equivalent Petri-net, that is, a transition is generated for each episode and alternative solution step; input places are generated for pre-conditions, conditions, and causes; output places are generated for post-conditions, and additional dummy transitions are generated for conditional or loop episodes. Places and transitions are labeled accordingly, and linked using arcs. Therefore, the execution of the episodes or an alternative flow in a scenario is modeled by firing a sequence of transitions in the resulting Petri-net.

Moreover, the *integration procedure* preserves the original properties and concurrency characteristics of the original Petri-nets into a consistent whole Petri-net *IPN*, that is, for any two Petri-nets *PN* and *IPN*: the fusion of common places between *PN* and *IPN*, or the substitution of a place or transition in *IPN* by *PN* do not introduce new arcs into the whole Petri-net *IPN*. New arcs might introduce new non-deterministic situations, and these are the main sources of synchronization problems.

4.5. Analyze

This activity checks both structural and behavioral aspects of scenarios and their equivalent Petri-nets by following the heuristics described in Table 2. Details about each *verification heuristic*, its defect indicators, detection methods, and fix recommendations are available in Appendix D.[7]

4.5.1. Static analysis

Ambiguity and incompleteness detection include:

Unambiguity Analysis: For each scenario, we read its title, goal, episodes and alternative flows; and identify defect indicators using lexical and syntactical strategies:

- *Dictionaries of frequently used ambiguous words*: For checking that a statement is free of *vague*, *optional*, *weak*, *quantification* and *non-minimal* words/phrases.
- *POS tagging*: For checking that a statement is free of *implicit* and *subjective* words or phrases.
- *Dependency parsing*: For checking that a statement does not contain *multiple* action-verbs or subjects.

Fig. 13a illustrates the use of these strategies for ambiguity detection. The following are examples of defect indicators pointed out by our

[7] Appendix D is available at https://bit.ly/2MpcMKQ.

Fig. 13. Evaluating unambiguity (a), completeness (b), and consistency (c) properties in the "Submit Order" scenario: The potential defect indicators and their detection methods.

approach in the "Submit Order" scenario; the underlined words stand for defect indicators in a statement:

- Dependency parsing ⇒ *Multiplicity* – Multiple action-verbs in: "3. The Customer enters her login information and selects the password reminder option".
- Dictionary ⇒ *Vagueness* – Vague words in: "2a1. The Broker System displays a login timeout page quickly".

Completeness Analysis: For each scenario, we read its title, goal, actors, resources, episodes, and alternative flows; traverse its related scenarios; derive a Petri-net; and identify defect indicators using lexical, syntactical, and graph traversal strategies:

- *Dictionary of frequently used ambiguous words*: For checking that the title and goal are free of *conjunction* and *disjunction* (and, or).
- *String searching*: For checking that (1) every actor and resource *participates* in at least one episode or alternative flow; (2) an included scenario *exists* within the set of scenarios;
- *Regular expressions*: For checking that a scenario statement is *conform* to the scenario writing guidelines.
- *Levenshtein's distance* (Levenshtein, 1966): For checking that (1) any two scenarios do not have *similar* titles. and (2) episode *coincidence* only takes place in different scenarios;
- *POS tagging*: For checking that the action-verb in the title, episode sentence, and alternative solution step sentence is described in its *infinitive* base form or *present* simple tense.
- *Dependency parsing*: For checking that (1) a statement is described by action-verbs, subjects, and objects; (2) a statement does not contain *multiple* action-verbs or subjects; (3) a *subject* mentioned in any episode sentence or alternative solution step sentence is an actor, a resource or the system;

- *Syntactical similarity* (compare action-verbs and objects): For checking that (1) the title *describes* the goal; (2) the title and goal are *unique*.
- *Breadth-first search*: For checking that exist at least a path between the *start place* (the scenario triggering) and every *final transition* (the last episode and the last solution step sentence of an alternative flow) of the derived Petri-net.

Fig. 13b illustrates the use of these strategies for incompleteness detection. The following are examples of defect indicators pointed out by our approach in the "Submit Order" scenario; the underlined words stand for defect indicators involving one or more statements:

- Syntactic similarity ⇒ *Conceptually Soundness* – The title does not describe the goal in: "TITLE: Submit Order" and "GOAL: Allow customers to find the best supplier for a given order".
- Dependency parsing ⇒ *Usefulness* – Undeclared actor in: "1. The user loads the login page".
- POS tagging and Dependency Parsing ⇒ *Simplicity* – Action-verb not in the third form in: "9. The Broker System broadcast the Order to the Suppliers".
- Dependency parsing ⇒ *Simplicity* – Missing [direct] object in: "12e The system exits gracefully".

4.5.2. Dynamic analysis

Static analysis is not useful for detecting *inconsistency* (Denger et al., 2005); thus, we transform and integrate related scenarios into a Petri-net *PN* and perform its *reachability analysis* (Murata, 1989). The reachability graph G can be used to examine:

- *Boundedness*: For checking the absence of overflowed resources, i.e., ∞ does not appear in any *node label*.
- *Liveness*: For checking the absence of: (1) never enabled *transitions*; and (2) paths to *deadlock*, i.e., every node M reachable from M_0 has adjacent arcs, except the nodes that finish the scenario.
- *Reversibility*: For checking that the model reaches its initial state again, i.e., a *live* and *bounded* ordinary Petri-net is reversible (Murata, 1989).

Other properties can be checked by also studying the topological structure of the Petri-net *PN*:

- *Persistence*: For checking the absence of non-deterministic execution paths, i.e., for any two transitions t_i and t_j that are adjacent to a marking M' in $G \rightarrow {}^\circ t_i \cap {}^\circ t_j = \varnothing$ in $PN \wedge t_i$ and t_j denote episodes.

Fig. 13c illustrates the use of *reachability analysis* for inconsistency detection. The following are examples of defect indicators pointed out by our approach in the integrated Petri-net (as in Fig. 12b) derived from the "Submit Order" scenario and its related scenarios; the underlined text stand for statements involved in potential defects:

- Boundedness ⇒ *Overflowed places* – "The post-conditions Local supplier has submitted a bid and International supplier has submitted a bid are overflowed" when transitions $T10$ and $T11$ are substituted by the Petri-nets corresponding to the referenced scenarios (SUPPLIERS).
- Liveness ⇒ *Path to deadlock* – "The Unhandled Alternative 8a leads to a path to deadlock".
- Liveness ⇒ *Never enabled transition* – "The exception solution 12e is never enabled".
- Persistence ⇒ *Non-deterministic execution path* – "There is a non-deterministic execution path in Episodes 10 and 11".

The*feedback generator* module can trace the defects reported from the Petri-nets analysis to defects in the equivalent scenarios, and simulate the sequence of episodes and alternative flows involved in them (as in Fig. 14).

State explosion issue is a serious problem when applying Petri-net analysis to large systems. We manage this issue by performing the reachability analysis of an *Integrated Petri-Net* in a compositional way. That is, we update the *procedure* 2 for integrating the Petri-nets derived from a given scenario and its related scenarios by: (1) Integrating the Petri-nets derived from the given scenario and its non-sequentially related scenarios; and (2) Adding the Petri-nets derived from the sequentially related scenarios into another set of Petri-nets. Given that the resulting Petri-nets preserve their original properties and concurrency characteristics; they can be analyzed separately.

Given the Petri-net derived from the "Submit Order" scenario (Fig. 12a), transitions $T10$ and $T11$ must be substituted by the Petri-nets derived from its non-sequentially related scenarios, such as depicted in Fig. 12b. However, transitions $T1e1$ and $T12$ are not substituted because they reference to sequentially related scenarios, and they can be analyzed separately.

4.6. Generate feedback

4.6.1. Findings

A *defect* report consists of the following sections:

<Property> - <Defect Category>: <Scenario Element> + <Indicator> + <Fix>, where: "*Property*" is the quality violated, "*Scenario Element*" is the section or a scenario statement where the defect occurs, "*Indicator*" describes the defect for fixing, and "*Fix*" gives a general recommendation for refactoring.

4.6.2. Simulation

Given a Petri-net *PN*, its reachability graph G and a marking M_n in G where a non-deterministic situation, a place to overflow, a deadlock, or a partial behavior occurs. We can reconstruct the *execution trace* in *PN* by traversing backward from M_n to the initial marking M_0 and finding the sequence of transitions that leads to M_n (Path: $M_0 \rightarrow t_0 \rightarrow M_1 \rightarrow t_1 \rightarrow \cdots \rightarrow M_{n-1} \rightarrow t_{n-1} \rightarrow M_n$). Thus, the validation of the general behavior of a set of related scenarios is based on the visual animation of an integrated Petri-net.

A token player tool can be used to simulate a partial behavior in *PN*. For instance, Fig. 14 depicts the execution paths (as a sequence of enabled-fired transitions) that lead to non-deterministic execution paths (a), places to overflow (b) and a deadlock (c) in the integrated Petri-net (as in Fig. 12b) derived from the "Submit Order" scenario and its related scenarios. Fig. 14d also depicts the never enabled transitions. Places and transitions highlighted in red boxes represent the trace or point where an erroneous situation occurs. To add semantics to the animations, we can label the transitions with their meanings in terms of scenarios, e.g., $T12$ ⇒ "PROCESS BID".

4.7. Fix

Given the feedback provided by the C&L, the requirements engineers are better prepared to review the scenarios and address their defects. Different actions may occur, including the elicitation of new facts and scenarios refactoring.

5. Evaluation

To evaluate the accuracy of the results produced by our analysis approach, we conducted a multi-case study with both industrial and academic projects. The guidelines of Runeson and Höst (Runeson and Höst, 2008) were largely followed for structuring our evaluation.

The evaluation reused data gathered from an earlier exploratory multi-case study (Sarmiento, 2016), and it was based on *Gold Standards* produced by the subjects and reviewed by the first author. The gold standards were used as a benchmark to measure the recall and precision of our automated analysis approach.

The evaluation addresses the following research questions:

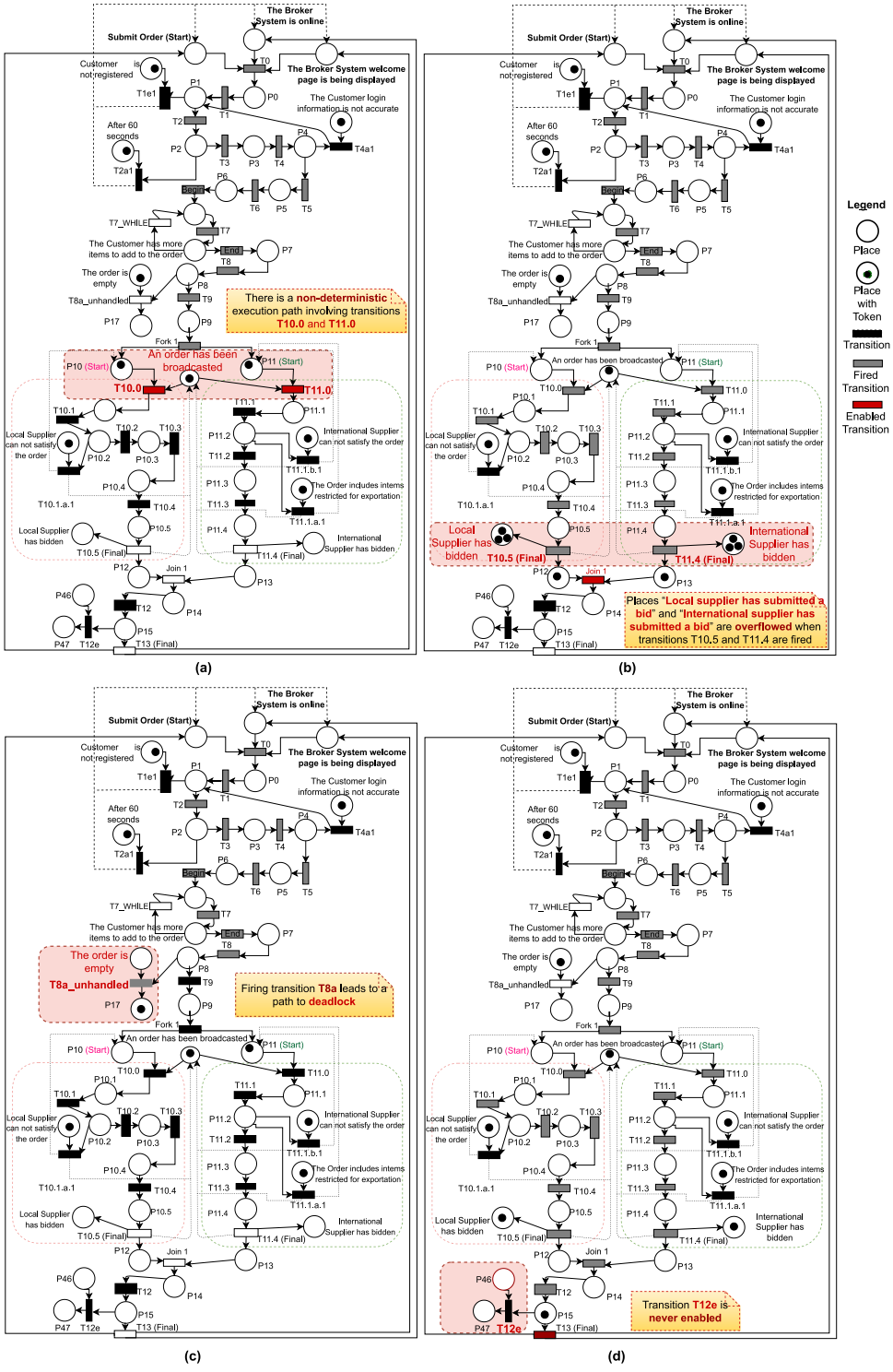


Fig. 14. Simulating the execution paths that lead to a non-deterministic situation (a), overflow (b), deadlock (c) and never enabled operation (d) in the integrated Petri-net (Fig. 12b) derived from the "Submit Order" scenario.

RQ1: Will the proposed automated analysis approach detect defects in SRSs in due time?

RQ2: Will the proposed automated analysis approach detect defects in SRSs accurately?

5.1. Preliminary evaluation

The reused data was based on the following design.

5.1.1. Case selection

We analyzed some projects in the literature (Ciemniewska and Jurkiewicz, 2007; Cox et al., 2003; Somé, 2010), that were representative enough for generalizing the findings of this evaluation. As *selection criteria*, we took into consideration the following parameters: (1) Access to a project in early stages of requirements specification and containing typical defects, (2) Ensure reasonable diversity of software domains, (3) Ensure the availability of the project SRS in the community, and (4) Ensure reasonable scale.

5.1.2. Subject selection

The subjects were 5 volunteers who played the role of Requirements Engineers. These volunteers were Master and Ph.D Computer Science researchers, and they have been working in the industry for the last 10 years. Particularly, they have been working with use cases (Cockburn, 2000) (Jacobson et al., 1992) and scenarios (do Prado Leite et al., 2000) for at least 5 years. To build a shared understanding of their tasks, they received training about the scenario language (Fig. 2) and its verification heuristics and defect indicators. With this background their task was to apply the verification heuristics to find defects.

5.1.3. Data collection procedure

We used a 2-step procedure to collect the data necessary for answering the research questions:

Collect Projects (Step 1): The researchers selected 4 projects specified as use case descriptions and publicly available in the literature, which were used to evaluate other proposals for use cases analysis (Somé, 2010; Cox et al., 2003; Ciemniewska and Jurkiewicz, 2007). Table 3 details the size of these projects according to the number of use cases and the number of statements described within them:

- *The Broker System* (Somé, 2010) consists of six use cases, whose goal is to allow customers to find the best supplier for a given order. These use cases were developed for evaluating an approach to formalize textual use cases through the Petri-nets formalism.
- *The ATM System* (Cox et al., 2003) consists of five use cases that describe the functionalities to produce a new cash point (ATM) for a bank. The authors introduced defects into the use cases to evaluate a manual inspection technique based on checklists.
- *The DLibra* and *Mobile News* (Ciemniewska and Jurkiewicz, 2007) are projects developed at Poznan University of Technology, for real customers from the university unit, industry or other organizations. *DLibra* consists of 15 use cases that describe the functionalities of a Web-based Customer Relationship Management system for managing the clients of a software for the creation of digital libraries. And, *Mobile News* consists of 15 use cases that describe the functionalities of a feed system for delivering the latest bulletins to mobile devices. These use cases contain industry-typical defects and were used to evaluate an automated approach for use cases verification.

In all cases, the use cases were redescribed using the general template showed in Fig. 2, without involvement from the subjects. These use cases are available in Appendix E.[8]

[8] Appendix E is available at https://bit.ly/3dDzk5x.

Table 3
Characteristics of the case studies.

	Broker System (Somé, 2010)	ATM (Cox et al., 2003)	DLibra (Ciemniewska and Jurkiewicz, 2007)	Mobile News (Ciemniewska and Jurkiewicz, 2007)
Num. of episodes/Steps	32	33	80	89
Num. of alternatives: / Solution steps	9	6	51	6
Num. of conditions/ Causes and pre-conditions	15	8	33	0
Num. of post-conditions	2	9	0	0
Total statements (length)	58	56	164	95
Num. of use cases	***6***	***5***	***15***	***15***

Elaborate the Gold Standard (Step 2): Whereas the analysis with case studies is comparative, we need to contrast the results obtained by our approach with another one. This Step was carried out by the subjects, and their task was to obtain a *Gold Standard* for each project case study. To ensure that all the subjects performed the analysis consistently, we carefully defined an abstract analysis protocol that was followed by each subject (as in Procedure 3).

Procedure 3 Protocol for Scenarios Analysis

Input: Set of scenarios $S = \{S_1, S_2, \dots, S_n\}$
Output: Analysis Report R
for each scenario $S_i \in S$ **do**
  for each statement $st \in S_i$ **do** ▷ Table 2
    Verify st against each *unambiguity* property
    Verify st against each *completeness* property
    Verify st against each *consistency* property
    if st does not fulfill some property **then**
      Add a *defect* in R: <Property> + <Scenario Statement> + <Defect Indicator>
  for each $S_j \in S$ **do** ▷ Table 2
    if S_j is related to S_i **then**
      Verify S_i and S_j against each *completeness* property
      Verify S_i and S_j against each *consistency* property
      if S_i and S_j do not fulfill some property **then**
        Add a *defect* in R: <Property> + <Scenario Statement in S_i > and <Scenario Statement in S_j > + <Defect Indicator>

We detail the process of creation of the gold standards for each case study as follows:

- *The Broker System*: This case was not used for use cases analysis. Then, all the subjects were allocated 1 h to perform independently the use cases analysis.
- *The ATM System*: Two subjects were allocated 1 h to review and validate the preliminary analysis results reported in Cox et al. (2003); then, they mapped the reported defects to the unambiguity and completeness properties defined in Table 2. Finally, they identified new defects.
- The *DLibra* and *Mobile News*: For each case, two subjects were allocated 3 h to review and validate the preliminary analysis results reported in Ciemniewska and Jurkiewicz (2007); then, they mapped the reported defects to the unambiguity and completeness properties defined in Table 2. Finally, they identified new defects.

Subsequently, the results of each subject were validated and amalgamated into a single gold standard. This process spent 1 h for each case, and the researchers (first author) acted as a moderator and coach. All discrepancies were discussed taking into account the background about scenarios quality (Table 2) and an agreement was reached.

5.2. The revised gold standard

Since the publication of the preliminary multi-case study (Sarmiento, 2016), the heuristics and their implementation into the C&L were revised, leading to the current version of our analysis approach. The trigger for the revision was the findings, by the first author, that the heuristics and the scenarios template parser implemented into the earlier version of the C&L could be improved.

Once the C&L evolved from Sarmiento (2016, 2019), Sarmiento-Calisaya et al. (2020) to the current version, we re-executed the analysis procedure for all cases. For each case, the classification accuracy metrics were computed as well as the execution time. By doing this, new results were showing a discrepancy with the original results (Sarmiento, 2016). That is, the evolved C&L: (1) was not flagging some defects previously detected, and (2) was flagging new defects. After re-examining the gold standards and the heuristics, it was found that the subjects failed in detecting some defects, i.e., some reported defects were not defects, and other ones were not detected in the original gold standards.

For example, the first version of the C&L and the experts classified the following statements as defective:

- *Quantifiability* – Unclear use of the quantification word "*all*" in: "*System deletes all returned messages from the database*". However, we believe that quantification words lead to ambiguity, if and only if, they are followed by a word that expresses vagueness; for instance, the word "required" in "*User fills all required data*".
- *Implicitly* – Use of the pronoun "*her*" in: "*The Customer provides her Credit Card information*". However, we believe that a pronoun is implicit, if and only if, there exist two antecedent nouns.
- *Simplicity* – Multiple action-verbs "*wants*" and "*change*" in: "*User wants to change their PIN*". However, we believe that a sentence containing a subordinate or coordinate clause (with its main verb) is not multiple, i.e., it can be described by an action-verb and a complement or modifier verb.

The new defects reported by the C&L were *vetted* by the researchers, and labeled as true positive for the tool and false negative for the subjects:

- *Persistence* – A non-deterministic situation because the scenarios "*Local Supplier Bid for Order*" and "*International Supplier Bid for Order*" are simultaneously enabled after performing the "*episode 9*" of the "*Submit Order*" scenario – The "Broker System".
- *Persistence* – A non-deterministic situation due to a GOTO and a scenario finishing statement described in the alternate solution steps 3.a.1 and 3.a.2 of the "*Delete Client*" scenario – The "DLibra".
- *Liveness* – A never enabled alternate solution due to the lack of a cause in the alternate 4.a of the "*Configure the server*" scenario – The "Mobile News".

This is always a threat when using subjects, even with a fair degree of experience. Upon the revision of the original gold standards, changes were made to deal with the identified problems, and new gold standards were built.

This is another example of the problem with gold standards in RE experiments. We agree with the observation made by Portugal et al. (2018), "*This is an example of Berry's argument over building gold standards (Berry, 2017, 2021); in particular, when the target of the gold standard is an identification regarding quality*". So, even with the threat inherent to the gold standard, for RE case studies, we believe the revised gold standards are an improvement over previous ones.

Table 4
Data about the revised gold standards for each case.

	Broker System	ATM System	DLibra	Mobile News	Total
Vagueness	–	–	1	5	6
Subjectiveness	–	–	–	–	–
Optionality	–	–	–	–	–
Weakness	–	–	8	–	8
Multiplicity	3	1	10	17	31
Implicitly	2	2	2	13	19
Quantifiability	–	–	4	–	4
Atomicity	–	–	1	–	1
Simplicity	6	6	39	25	76
Uniformity	–	2	–	3	5
Usefulness	6	2	–	3	11
Conceptually soundness	3	–	–	–	3
Integrity	–	–	–	–	–
Coherency	–	–	–	–	–
Uniqueness	2	–	–	–	2
Persistence	3	2	1	–	6
Boundedness	–	–	–	–	–
Liveness	–	2	–	3	5
Reversibility	–	–	–	–	–
Total relevant answers	**25**	**17**	**66**	**69**	**177**

Table 4 summarizes the data about each case, i.e., the number of defects and their classification in the revised gold standards. The gold standards are shown in Appendix E.[8]

We can see that the scenarios contain defect indicators that negatively impact their quality, mainly their Simplicity (e.g., complex sentences, action-verbs in the incorrect form and missing objects), Uniformity (e.g., alternatives without their relevant parts), Multiplicity (e.g., multiple action-verbs and subjects), Implicit (e.g., implicit pronouns), Usefulness (e.g., undeclared actors, lack of actor/system in sentences and too short or too long sequence of episodes), Persistence (e.g., simultaneously enabled operations), and Liveness (e.g., never enabled operations).

We have a large number of defects related to weakness, multiplicity, and simplicity because many sentences are described by more than one action-verb or an action-verb in passive form, i.e., some defects related to simplicity are also reported as defects related to multiplicity or weakness.

Following Berry's suggestions about *Gold Standard Construction* (Berry, 2021), for each subject, we collected: (1) The time spent analyzing each case, and (2) The recall and precision with respect to the gold standards after a consensus was reached. The averages of the data collected from the subjects and researchers are shown in Table 5. This table shows the average values as if they were about a single average subject. The average of the subjects' recall and precision are taken as the *Humanly Achievable Recall (HAR)* and *Humanly Achievable Precision (HAP)*, and serve as the target recall and precision to be achieved by any proposed substitute for scenarios analysis manually.

5.3. Case studies with the revised gold standards

Given the data collection performed earlier, each project was stored in the *C&L*.

Our *goal* with the case studies is to show the usefulness of our automated approach to detect defects in scenarios with close to 100% recall and higher precision. That is, the C&L must achieve at least the *HAR* and *HAP*. In line with Berry's notion of a hairy-RE-task tool (Berry, 2021).

Table 5
Data about averages in the construction of the gold standards for each case.

	Broker System	ATM System	DLibra	Mobile News	Total
Minutes to analyze manually and independently	60	60	180	180	480
Minutes to validate and amalgamate manually	60	60	60	60	240
Minutes to produce the initial Gold Standard	120	120	240	240	720
Minutes to review the initial Gold Standard	60	60	120	120	360
Minutes to produce the Revised Gold Standard	180	180	360	360	1080
Recall - HAR, $R_{aveSubject}$ for Unambiguity	0.6	1	1	0.91	0.88
Precision - HAP, $R_{aveSubject}$ for Unambiguity	0.43	0.33	0.68	0.58	0.51
Recall - HAR, $R_{aveSubject}$ for Completeness	1	1	0.88	0.84	0.93
Precision - HAP, $R_{aveSubject}$ for Completeness	0.94	0.91	0.9	0.9	0.91
Recall - HAR, $R_{aveSubject}$ for Consistency	0.67	0.5	0.67	0	0.46
Precision - HAP, $R_{aveSubject}$ for Consistency	1	1	1	0	0.75
Recall - HAR, $R_{aveSubject}$ - Overall	0.88	0.88	0.91	0.91	0.9
Precision - HAP, $R_{aveSubject}$ - Overall	0.81	0.68	0.79	0.79	0.77

Table 6
Standard data from comparing the C&L output with the revised gold standards.

	Broker System	ATM System	DLibra	Mobile News	Total
Relevant answers found (TP) for unambiguity	5	3	24	34	66
Irrelevant answers found (FP) for unambiguity	2	0	1	3	6
R_{raw} for unambiguity	1	1	0.96	0.97	0.98
P_{raw} for unambiguity	0.71	1	0.96	0.89	0.89
Relevant answers found (TP) for completeness	17	10	40	31	98
Irrelevant answers found (FP) for completeness	0	2	0	1	3
R_{raw} for completeness	1	1	1	1	1
P_{raw} for completeness	1	0.83	1	097	0.95
Relevant answers found (TP) for consistency	3	4	1	3	11
Irrelevant answers found (FP) for consistency	0	0	0	0	0
R_{raw} for Consistency	1	1	1	1	1
P_{raw} for Consistency	1	1	1	1	1
Relevant answers found (TP) - Overall	25	17	65	68	175
Irrelevant answers found (FP) - Overall	2	2	1	4	9
R_{raw} - Overall	1	1	0.98	0.99	0.99
P_{raw} - Overall	0.93	0.89	0.98	0.95	0.94

5.3.1. Analysis procedure

Our analysis procedure leads to the results necessary for answering the research questions:

Run the C&L Tool: The analysis approach implemented in the C&L tool (Sarmiento-Calisaya et al., 2020).

Calculate the Execution Time: During the evaluation of the case studies, time statistics were gathered to assess the performance of the developed solution. Answer the RQ1. ***Calculate the Precision***: Measure the rate of correct defects found by the approach (True Positive – *TP*) in contrast to the amount of incorrect detections (False Positive – *FP*). Answer the RQ2.

$$Precision(P) = \frac{TP}{TP+FP}$$

Calculate the Recall: Measure the rate of correct defects identified by the approach (True Positive – *TP*) in contrast to the number of missed defects (False Negative – *FN*). Answer the RQ2.

$$Recall(R) = \frac{TP}{TP+FN}$$

5.3.2. Running the C&L

Applying the C&L analysis on each case,[8] a list of defects was produced per each scenario. The first author examined the 181 found answers by the C&L against the revised gold standards, and computed the recall and precision of the tool, R_{raw} and P_{raw}.

The data from the comparison of the C&L output with the revised gold standards are shown in Table 6. The C&L was accurate, making only a few mistakes, showing 9 irrelevant answers – false positives and missing 2 relevant answers – false negatives. Thus, the R_{raw} and P_{raw} beat the *HAR* and *HAP*, respectively, in all cases.

5.3.3. Results and discussion

In the following, we answer the research questions of the case studies.

RQ1: Will the proposed automated analysis approach detect defects in SRSs in due time?

The processing times were 8.05, 8.76, 18.91, and 19.45 s, for the Broker System, ATM System, DLibra, and Mobile News cases. The "*Mobile News*" case took more time than the "*DLibra*" because the length (number of words) of the statements of the use cases in the "*Mobile News*" case is larger than the "*DLibra*" case; so we assume that the NLP POS tagging and Parsing tasks took more time.

Answer to RQ1: The execution of C&L in the case studies demanded a very short time.

Studies about industry projects revealed that the number of scenarios is typically limited (Somé, 2010), i.e., large projects usually demand around 80 scenarios (Rago et al., 2016). Moreover, scenarios usually have 3–9 steps in the main flow, and these scenarios had more than 1 alternative flow of 1–4 steps each (Alchimowicz et al., 2008; Cox et al., 2003).

The industrial and academic projects used as input data in our cases have characteristics of typical scenario/use-case usage (Alchimowicz et al., 2008; Cox et al., 2003; Somé, 2010); therefore, scaling to much larger projects should not be an issue.

RQ2: Will the proposed automated analysis approach detect defects in SRSs accurately?

To evaluate the accuracy of the results achieved by the C&L on each case study, we measured the *recall* and *precision* of the tool, R_{raw} and P_{raw}. The data collected from the case studies execution are shown in Tables 6 and 7.

Table 7
Analysis of unambiguity, completeness, consistency, and correctness using the C&L.

Unambiguity	Broker System					ATM System					DLibra					MobileNews				
	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}
Vagueness	0	1	0	1	0	0	0	0	–	–	1	0	0	1	1	5	1	0	1	0.83
Subjectiveness	0	1	0	1	0	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Optionality	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Weakness	0	0	0	–	–	0	0	0	–	–	8	0	0	1	1	0	0	0	–	–
Multiplicity	3	0	0	1	1	1	0	0	1	1	9	1	1	0.9	0.9	16	3	1	0.94	0.84
Implicitly	2	0	0	1	1	2	0	0	1	1	2	0	0	1	1	13	0	0	1	1
Quantifiability	0	0	0	–	–	0	0	0	–	–	4	0	0	1	1	0	0	0	–	–
Total	**5**	**2**	**0**	**1**	**0.71**	**3**	**0**	**0**	**1**	**1**	**24**	**0**	**1**	**0.96**	**0.96**	**34**	**1**	**1**	**0.97**	**0.89**
Completeness	Broker System					ATM System					DLibra					MobileNews				
	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}
Atomicity	0	0	0	–	–	0	0	0	–	–	1	0	0	1	1	0	0	0	–	–
Simplicity	6	0	0	1	1	6	0	0	1	1	39	0	0	1	1	25	1	0	1	0.96
Uniformity	0	0	0	–	–	2		0	1	1	0	0	0	–	–	3	0	0	1	1
Usefulness	6	0	0	1	1	2		0	1	1	0	0	0	–	–	3	0	0	1	1
Conceptually Soundness	3	0	0	1	1	0	2	0	–	0	0	0	0	–	–	0	0	0	–	–
Integrity	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Coherency	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Uniqueness	2	0	0	1	1	0	0	0	–	–	0	0	0	–	–	0	0	0		
Total	**17**	**0**	**0**	**1**	**1**	**10**	**2**	**0**	**1**	**0.83**	**40**	**0**	**0**	**1**	**1**	**31**	**1**	**0**	**1**	**0.97**
Consistency	Broker System					ATM System					DLibra					MobileNews				
	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}	TP	FP	FN	R_{raw}	P_{raw}
Persistence	3	0	0	1	1	2	0	0	1	1	1	0	0	1	1	0	0	0	–	–
Boundedness	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Liveness	0	0	0	–	–	2	0	0	1	1	0	0	0	–	–	3	0	0	1	1
Reversibility	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–	0	0	0	–	–
Total	**3**	**0**	**0**	**1**	**1**	**4**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**3**	**0**	**0**	**1**	**1**
Correctness	Broker System					ATM System					DLibra					MobileNews				
	R_{raw}		P_{raw}			R_{raw}		P_{raw}			R_{raw}		P_{raw}			R_{raw}		P_{raw}		
	1		**0.93**			**1**		**0.89**			**0.98**		**0.98**			**0.99**		**0.95**		

Unambiguity Analysis: Overall, the C&L produced results with a recall and precision greater than 96% and 71%.

In the "*DLibra*" and "*Mobile News*" cases, the C&L achieved a recall of 96% and 97%, respectively, due to the imprecision of the strategy for multiplicity detection, which is based on NLP dependency parsing. For example, in the following sentence, the action-verbs "*checks*" and "*inserts*" were not labeled as multiplicity indicators, which is inconsistent with the requirements engineer's decision:

"*System checks if a group with the given name has not been already defined and if so, inserts the name of a new group into a database*"

In the "*Broker System*" case, the C&L achieved a low precision of 71% due to the imprecision of the strategies for vagueness and subjectiveness detection, which are based on NLP POS tagging. For example, in the following sentence, the word "*provided*" was labeled as a vague word, which is inconsistent with the requirements engineers' decision:

"*The Broker System checks the provided login information*"

We can improve the precision in detecting vague, multiple, and implicit words by splitting a statement into multiple sentences and integrating a *NLP coreference resolution* annotator or an anaphoric ambiguity detection heuristic (Rosadini et al., 2017) into the C&L.

Completeness Analysis: Overall, the C&L produced results with proper recall and precision greater than 83%.

In the "*ATM System*" case, the C&L achieved a low precision of 83% due to the imprecision of the '*Syntactic Similarity*" heuristic. We noticed that some titles and their goals were described by some words (nouns or verbs) and their synonyms. For example, in the following sentences, the words "*cash*" and "*money*" were understood as different terms:

"*Withdraw cash*" and "*User wants to withdraw money*".

We can improve the precision of the "syntactic similarity" heuristic by bringing more semantics to it, e.g., we can include the synonyms (from WordNet database)[9] of the verbs and nouns for comparison – *semantic similarity*.

Consistency Analysis: Overall, the C&L was very accurate in detecting non-deterministic situations, paths to deadlock, and never enabled operations.

In the case of *deadlocks* and *overflowed resources*, our approach appoints *information* messages instead of *warning* or *mistake*. Petri-net analysis tools cannot notice that an overflowed place representing a scenario/use-case post-condition is not a defect. Similarly, a path to deadlock raised from an alternative/exception is a non-conclusive defect. Thus, they are not considered for recall and precision.

To improve the precision of these heuristics, we can redescribe scenario/use-case steps in terms of pre-conditions and post-conditions, such as proposed by Lee et al. (1998). However, this strategy requires manual intervention.

Correctness Analysis: Correctness is positively contributed by unambiguity, completeness, and consistency; thus, we can aggregate the results produced by related qualities. Overall, our approach produced reasonable results, with recall and precision greater than 98% and 89%. Aggregated values of the above accuracy metrics are as in Table 7.

Answer to RQ2: The C&L was accurate, making only a few mistakes. The overall *recall* and *precision* were 99% and 94%. It means that, for those defects correctly identified, our approach can significantly improve the quality of scenarios. The resulting quality analysis shows promising results that indicate a high potential for successful further improvements.

Although these projects describe some abstract systems, they show the typical defects (Alchimowicz et al., 2008) of the industrial requirements specifications, and, they can be used in different ways by both researchers and practitioners.

[9] Wordnet is available at https://wordnet.princeton.edu/.

We believe that the promising results motivate the use of the C&L tool over doing the analysis manually. However, it is important to investigate practitioners' perceptions of the benefits of our automated approach.

5.4. Threats to validity

Regarding ***Construct Validity***, the main remark about construct validity has to do with what it means defects that characterize inadequate quality of scenarios or use case descriptions. Before the gold standards construction and case studies execution, the use case descriptions in the projects were redescribed using the general template showed in Fig. 2 and stored into the *C&L*, i.e., a manual effort was needed to copy and paste the original textual use case descriptions. In this process, some issues may be introduced; however, this is not considered harmful because the proposed general template and the C&L support the templates used in these projects, ensuring wide applicability. Subsequently, we classified defects severity as *mistake*, *warning* or *information*. This can potentially relax the definition of scenario quality. Therefore, defects labeled as information were not considered for recall and precision.

The adopted recall and precision metrics are means for measuring the performance of our approach; however, these metrics have no consideration for the *relevance* of the defects found. More experimentation may reveal nuances due to different contexts.

In the case of ***Internal Validity***, we reduced the subjectivity in the construction of the gold standards. The requirements engineers had similar degrees of background about scenarios, use case templates, and requirements quality; and we provided the verification heuristics and their defect indicators. However, the volunteers might have been influenced by their personal interpretations of defect indicators.

It is interesting to note that in some cases, the experts failed to detect defects; that is, human oracles are not necessarily right. Thus, the gold standard elaboration depends on the context and the subjects involved in such a situation (Portugal et al., 2018).

In the case of ***External Validity***, generalizability is always a concern. With respect to the use of researchers as experimental subjects, we consider that the results can be generalized to industry practitioners because: (1) the researchers have a fair industrial background in requirements engineering practices, (2) the projects describe different domains and use different scenario templates, and (3) the projects contain typical defects of the industrial requirements specifications. In any case, we are aware that more study is required to ensure that our approach remains effective with diverse subjects and in other application domains and sizes.

6. Related work

Most of the efforts are targeted towards the definition of writing guidelines, quality models, a taxonomy of common defects indicators, checklists, or metrics to quantify the presence of evidence (or not) of quality properties in requirements. These efforts are either towards *high-level requirements* descriptions, or detailed descriptions of system functionalities – *scenarios*.

6.1. Analysis of high-level requirements

Most of the efforts are targeted towards the automatic detection of ambiguity indicators in *high-level requirements* statements (Arora et al., 2015; Femmer et al., 2017; Lami et al., 2004; Wilson et al., 1997; Rosadini et al., 2017; Tjong and Berry, 2013) or *user stories* (Lucassen et al., 2015). These approaches defined quality properties and indicators (inspired in Wilson et al., 1997 and IEEE Computer Society, 1998) and, developed tools that implement static analysis strategies (Dictionaries or POS tagging) for searching ambiguity indicators within individual requirements. Rosadini et al. (2017) proposed a set of patterns for detecting ambiguity indicators, and they validated their approach on industrial requirements annotated by domain experts.

A few works Arora et al. (2015) and Lucassen et al. (2015) were proposed for identifying defect indicators that negatively impact completeness (mainly, conformance to requirements templates). Arora et al. (2015) check ambiguity indicators within requirements and their conformance to *Rupp* and *EARS* templates. Lucassen et al. (2015) check the conformance to a *User Story* template, and some properties (e.g., dependency and duplicity) involving relationships among user stories, characterizing and formalizing each relationship via first-order logic.

In Table 8, based on a set of features (e.g., Are relationships among requirements considered for analysis? Is it tool-supported? Does it detect unambiguity defects?), we summarize the differences between these approaches.

The proposed approaches are also useful to evaluate scenarios, since ambiguity indicators are present in statements within scenario sections such as demonstrated in Femmer et al. (2017) and Lami et al. (2004).

6.2. Analysis of scenario-based requirements

Most of the existing approaches are focused on checking the *structural* properties related to completeness, i.e., checking the conformance to scenarios/use-case templates (Chu et al., 2017; Cockburn, 2000; Kruchten, 2003; do Prado Leite et al., 2000; Somé, 2010; Yue et al., 2013; Wang et al., 2020) or guideline writing rules (Achour et al., 1999; Chu et al., 2017; Cox and Phalp, 2000; Yue et al., 2013; Wang et al., 2020), and the coherence between scenario components. Inspection techniques (Anda et al., 2009; do Prado Leite et al., 2005, 2000; Phalp et al., 2007) with checklists were proposed for measuring the compliance (content and style) of scenarios with writing guideline suggestions. A few works Anda et al. (2009) and do Prado Leite et al. (2000) checked properties involving a set of scenarios (e.g., sequential relationships like use/extend use case relationships). In these approaches, scenarios/use-cases descriptions are manually inspected and related scenarios are explicitly referenced (e.g., using special keywords). Phalp et al. (2007) indicate that when checklists are used in manual inspections, it is mostly syntax errors that will be discovered because they are easier to find than semantic ones. Some works (Ciemniewska and Jurkiewicz, 2007; Liu et al., 2014; Sinha et al., 2010; Rago et al., 2016; Wang et al., 2020) present tool-supported approaches based on NLP strategies, other approaches (do Prado Leite et al., 2000) provide insights for further automation.

To evaluate the *behavioral* properties related to consistency or correctness of textual scenarios, it is necessary to execute or simulate them. Several studies have shown the importance of formalizing them through restricted-form of use case descriptions (Chu et al., 2017; Sinha et al., 2010; Sinning et al., 2009; Somé, 2010; Yue et al., 2013). Other approaches translate use cases into formal representations and analyze the resulting models (Chu et al., 2017; Sinning et al., 2009; Denger et al., 2005; Cheung et al., 2006; Lee et al., 1998; Somé, 2010; Lee et al., 2001).

6.2.1. Static analysis of scenarios

NLP-based approaches (Ciemniewska and Jurkiewicz, 2007; Liu et al., 2014; Sinha et al., 2010; Rago et al., 2016; Wang et al., 2020) get knowledge and relevant information from use case events (steps in the main flow and alternatives) and verify that are correctly written and free of duplicity. With the aid of *POS tagging* and *dependency parsing*, they extract the *subject*, *action-verb* and *object* of a statement; and appoint incompleteness or not. To improve the accuracy of the parsers, domain knowledge is needed to introduce information and train the parser by providing annotated data. A few works (Ciemniewska and Jurkiewicz, 2007; Liu et al., 2014; Rago et al., 2016) proposed heuristics for finding defects involving a set of use cases, e.g., dependency and duplicity.

Table 8
Summary of analysis approaches for high-level requirements.

	Wilson et al. (1997)	Lami et al. (2004)	Tjong and Berry (2013)	Arora et al. (2015)	Lucassen et al. (2015)	Femmer et al. (2017)	Rosadini et al. (2017)
Requirement representation	Natural Language	Natural Language	Natural Language	Rupp's; EARS;	User Story;	Natural Language	Natural Language
Analysis technique	Dictionary of Indicators;	Dictionary of Indicators; NLP;	Dictionary of Indicators; NLP;	Dictionary of Indicators; NLP;	Dictionary of Indicators; NLP;	Dictionary of Indicators; NLP;	Dictionary of Indicators; NLP;
Relationships					✓		
Tool-supported	ARM	QUARS	SREE	✓	AQUSA	SMELLA	✓
Unambiguity	✓	✓	Yes	✓	✓	✓	✓
Completeness			Partial	✓	✓	Partial	Partial
Consistency		Partial			Partial		
Correctness							
Scenarios?	✓	✓	Yes	✓	Partial	✓	✓

Ciemniewska and Jurkiewicz (2007) proposed heuristics based on NLP dependency parsing to extract relevant information from use case steps (main flow), and then detect *duplicity* by comparing the steps of two use cases.

Rago et al. (2016) proposed an approach that uses NLP to convert use case steps (main and alternative flow) into an abstract representation that consists of sequences of semantic actions, and then detect possible *duplication* by sequence matching (Pairwise Sequence Alignment) algorithms. They achieved a low precision due to the number of false positives, i.e., it is hard to differentiate between real duplication and very similar use case scenarios.

Liu et al. (2014) proposed a heuristic for finding relationships among use cases by constructing Deterministic Finite-State Automaton (DFA) from the behavior in individual use cases and composing them into a whole use cases graph, then, traversing this graph, it is possible to find potential *missing use cases* or whether a certain *pre-condition is satisfied by certain post-condition* (dependency). However, to find the relationships among use cases, an active learning strategy is used, i.e., they discover missing scenarios by generating questions to users – *user intervention*.

To evaluate the coherency between scenarios/use-case descriptions and other requirements artifacts (e.g., domain models), some approaches (do Prado Leite et al., 2000; Sinha et al., 2010; Wang et al., 2020) check the consistency, traceability, and change propagation between models. do Prado Leite et al. (2000) and Sinha et al. (2010) check that the vocabulary (lexicon) or domain model of the application is consistently used in scenarios/use-case descriptions, respectively. On the other hand, Wang et al. (2020) automatically check if the domain model includes all the entities mentioned in the use cases; thus, they identify defects in domain models and not in use cases.

6.2.2. Dynamic analysis of scenarios

In Sinning et al. (2009), Denger et al. (2005), Lee et al. (1998) and Somé (2010) are presented systematic procedures to convert use cases into formal representations. The resulting models can be analyzed using available tools and detect erroneous situations raised from the complex relationships among scenarios.

Lee et al. (1998) propose a systematic procedure to formalize use cases, by mapping use case descriptions into *Constraint-based Modular Petri-Nets* (CMPNs), allowing the analysis of use cases. Use cases are considered as a collection of interacting and concurrently executing units of system functionalities. They made use of Petri-nets analysis to detect *non-determinism*, *isolated subnets*, *never enabled transitions* and *deadlocks*. This approach manages the state explosion issue of the Petri-nets by dividing a CMPN into a set of slices. However, to facilitate the transformation, intermediate models (*action-condition tables*) are created, use cases are described with a formal definition of pre-conditions and post-conditions, and they use a non-standard use case model without alternative/exception flows.

Denger et al. (2005) present an integrated approach for achieving high quality in use cases that combines use case creation guidelines, inspection, and simulation in a systematic way. They base their approach on a defect classification for use cases, which enables the requirements engineer to focus the different techniques on different types of defects. They showed that guidelines are valuable for the prevention of structural and syntactic defects, and inspection is suitable for detecting subtle logical defects. Simulation is integrated so that consistency and correctness defects resulting from the *interference* between use cases can be efficiently detected; to this goal, use cases are mapped into *Statecharts*, and several statecharts can be simultaneously simulated.

Sinning et al. (2009) propose a syntax definition that formalizes the sequencing of use case steps and their types; then, a formal semantics based on *Labeled Transition System (LTS)* is proposed for use case models containing extend and include UML relationships. A use case model is mapped to UC-LTS by generating UC-LTSs from use case descriptions (steps and extensions) and, merging the UC-LTSs representing the various entailed use cases. They developed the use case analyzer tool to automatically detect *livelocks*. They also propose a method for verifying the refinement of use case models, namely checking their equivalence and deterministic reduction. Most of the checks focus on the global properties of use cases and their sequential relationships (precedence).

Somé (2010) proposes an approach to formalize textual use cases via *Reactive Petri-Nets*. They provided an algorithm for the generation of reactive Petri-nets from use cases, which are described using a formal syntax, and taking into account the *include* and *extend* UML relationships and the sequencing constraints defined in their *pre/post-conditions*. The constructed Petri-net can be used to evaluate *incomplete behavior*, *temporal constraints* and *non-determinism*. However, this approach does not deal with communication between concurrent use cases and the non-explicit relationships among use cases.

6.3. Comparison with related work

In Table 9, based on a set of features necessary for the automatic analysis (static and dynamic) of scenarios, we summarize the differences between our approach and prominent related work. For each approach, we evaluate: What scenario representation is used? Is there a syntax for scenarios? What analysis technique is used? Are the relationships among internal components of scenarios considered (e.g., actors, steps, extensions)? Are the relationships among scenarios considered for analysis? Are the non-explicit relationships among scenarios considered for analysis? Are related scenarios integrated for analysis? Is the state explosion issue of reachability analysis managed? Is it tool-supported? Does it detect unambiguity defects? Does it detect completeness defects? Does it detect consistency defects? Does it detect correctness defects? For instance, the related work does not make available automated tools. This shortcoming significantly hinders their applicability.

Table 9
Summary and comparison of the scenario/use-case analysis approaches.

	do Prado Leite et al. (2000)	Ciemniewska and Jurkiewicz (2007)	Liu et al. (2014)	Lee et al. (1998)	Denger et al. (2005)	Sinha et al. (2010)	Sinning et al. (2009)	Somé (2010)	Our approach
Scenario representation	Scenario	Use Case	Use Case; Activity Diagram	Use Case; Action-Condition table	Use Case	Use Case diagram;	Use Case; Use Case diagram	Use Case; Use Case diagram	Scenario/Use-Case
Syntax for scenarios	✓	Partial	Partial		✓	Use Case;	✓	✓	Partial
Analysis technique	Checklist; Heuristics	Heuristics; NLP	NLP; Deterministic Finite-State Automaton	Constraints-based Modular Petri-Net;	Checklist; Statechart	Yes	LTS;	Reactive Petri-Net;	Regular Expression; NLP; Place/ Transition Petri - net
Relationships among internal components	✓	✓	Partial		Partial	✓	✓		✓
Relationships among scenarios	✓	Partial	Partial	✓	Partial	Partial	✓	Partial	✓
Non-explicit relationships among scenarios	Manual	Partial	Partial	✓					✓
Integration of related scenarios for analysis				✓	Partial		Partial		✓
Tool-supported		✓	✓			✓	Partial	✓	✓
State explosion	–	–	–	Slices					Bottom-up
Unambiguity	Partial	✓	Partial	Partial	Partial	Partial	Partial	Partial	✓
Completeness	✓	✓	Partial	✓	Partial	✓	Partial	Partial	✓
Consistency	Partial				Partial	Partial	Partial	Partial	✓
Correctness	Partial				Partial	Partial	Partial	Partial	Partial

In summary, our approach: (1) supports most of the proposed scenarios/use-case templates based on Cockburn (2000) Jacobson et al. (1992) and do Prado Leite et al. (2000); (2) provides powerful characteristics to deal with modularity, i.e., considers relationships among scenarios; (3) takes into consideration the results achieved by research about high-level requirements and user stories analysis; (4) models the quality properties of scenarios and their impacts; (5) lists a set of feasible verification heuristics that check structural and behavioral aspects of scenarios; (6) demonstrates the feasibility of our proposal by implementing the proposed heuristics and methods into a prototype tool (Sarmiento-Calisaya et al., 2020); and (7) improves the construction of more consistent and correct scenarios for test generation. Finally, automation reduces manual effort.

7. Conclusion

In this work, we introduced a tool-supported approach to analyze the structural and behavioral aspects of natural language-based scenarios. To enable the automatic analysis of scenarios, we benefit from the usability of textual scenarios and the availability of static and dynamic analysis strategies and tools such as the Stanford CoreNLP (Manning et al., 2014) and the PIPE2 (Dingle et al., 2009) Petri-net editor.

To extract structural and behavioral information from scenario specifications, we rely upon a general template that supports most of the existing scenarios/use-case templates, and an advanced NLP solution for role extraction (subject-verb-object). Then, we translated the scenarios into Petri-nets, which preserve the original properties of the scenarios. Finally, we implemented rule-based verification heuristics to check the unambiguity, completeness, consistency and correctness properties of scenarios.

We have evaluated our approach with four publicly available projects with promising results, a recall greater than 98% and precision greater than 89%. We also demonstrated that it is applicable to real projects involving scenarios with industry profiles. However, we are aware that more experimentation is needed to evaluate the *relevance* of the results.

The major *limitation* of our approach is that the transformation procedure from scenarios into Petri-nets is sensitive to the correct syntax of scenarios. This means that the transformation works well if the input scenarios are described through the common sections in scenario/use-case templates (as in Fig. 2) and putting the correct keywords (IF-THEN, DO-WHILE, so on) on statements. It is our assumption that the use of templates and keywords is well accepted by the stakeholders in the RE process. However, further empirical research is needed to corroborate this assumption. In the current version of our analysis approach, the input projects – *use case descriptions* used in the evaluation were not polished before.

We foresee different avenues for *future work*. They include improving heuristics, reusing other sources for consistency checking, improving the tool, and integrating change impact analysis. As for new heuristics, it would be proper to investigate other properties like traceability, modifiability and testability, as well as to explore anaphora (or pronouns) and coordination (and, or), as well as ambiguity resolution (Rosadini et al., 2017). The usage of previous knowledge such as WordNet, lexicon, glossary or domain model of the application, may be a way of bringing more semantics to the analysis. Our approach may be used with *incremental* development strategies and it also contributes to a better understanding of scenarios evolution and their *impacts*.

This work can also be extended to generate other artifacts such as domain models and test cases. For instance, the initial results of an approach to automatically traverse a Petri-net and its reachability graph to generate test scenarios based on path analysis strategies have been published in Sarmiento et al. (2016).

CRediT authorship contribution statement

Edgar Sarmiento-Calisaya: Conceptualization, Methodology, Software, Resources, Writing – original draft. **Julio Cesar Sampaio do**

Prado Leite: Conceptualization, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://github.com/edgarsc22/WACeL-Java.

References

Abad, Z.S.H., Noaeen, M., Ruhe, G., 2016. Requirements engineering visualization: A systematic literature review. In: 2016 IEEE 24th International Requirements Engineering Conference. RE, IEEE, pp. 6–15.

Achour, C.B., Rolland, C., Maiden, N.A.M., Souveyet, C., 1999. Guiding use case authoring results of an empirical study. In: Proceedings of the IEEE International Symposium on Requirements Engineering (Cat. No. PR00188). pp. 36–43. http://dx.doi.org/10.1109/ISRE.1999.777983.

Alchimowicz, B., Jurkiewicz, J., Nawrocki, J., Ochodek, M., 2008. Towards use-cases benchmark. In: CEE-SET-2008: Software Engineering Techniques, Vol. 4980. Springer, Berlin, Heidelberg, pp. 20–33. http://dx.doi.org/10.1007/978-3-642-22386-0_2.

Anda, B., Hansen, K., Sand, G., 2009. An investigation of use case quality in a large safety-critical software development project. Inf. Softw. Technol. 51, 1699–1711. http://dx.doi.org/10.1016/j.infsof.2009.04.005.

Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F., 2015. Automated checking of conformance to requirements templates using natural language processing. IEEE Trans. Softw. Eng. 41 (10), 944–968. http://dx.doi.org/10.1109/TSE.2015.2428709.

Berry, D.M., 2017. Evaluation of tools for hairy requirements and software en-gineering tasks. In: 2017 IEEE 25th International Requirements Engineering Conference Workshops. REW, pp. 284–291.

Berry, D.M., 2021. Empirical evaluation of tools for hairy requirements engineering tasks. Empir. Softw. Eng. 26 (6), 111.

Berry, D., Gacitua, R., Sawyer, P., Tjong, S.F., 2012. The case for dumb requirements engineering tools. In: Requirements Engineering: Foundation for Software Quality. Srpinger, Berlin, Heidelberg, pp. 211–2147. http://dx.doi.org/10.1007/978-3-642-28714-5_18.

Boehm, B.W., 1979. Guidelines for verifying and validating software requirements and design specifications. In: Proceedings of the European Conference Applied Information Technology. pp. 711–719.

Cambria, E., White, B., 2014. Jumping NLP curves: A review of natural language processing research. IEEE Comput. Intel. Mag. 9 (2), 48–57. http://dx.doi.org/10.1109/MCI.2014.2307227.

Cheung, K.S., Cheung, T.Y., Chow, K.O., 2006. A Petri-net-based synthesis methodology for use-case-driven system design. J. Syst. Softw. 79 (6), 772–790. http://dx.doi.org/10.1016/j.jss.2005.06.018.

Chu, M.-H., Dang, D.-H., Nguyen, N.-B., Le, M.-D., Nguyen, T.-H., 2017. Towards precise specification of use case for model-driven development. In: Proceedings of the Eighth International Symposium on Information and Communication Technology. In: SoICT 2017, ACM, pp. 401–408. http://dx.doi.org/10.1145/3155133.3155194.

Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., 2000. Non-Functional Requirements in Software Engineering. Springer, US.

Ciemniewska, A., Jurkiewicz, J., 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.

Cockburn, A., 2000. Writing Effective Use Cases, first ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Condori-Fernandez, N., España, S., Sikkel, K., Daneva, M., Gonzales, A., 2014. Analyzing the effect of the collaborative interactions on performance of requirements validation. In: Salinesi, C., de Weerd, I.V. (Eds.), In: Lecture Notes in Computer Science, vol. 8396, Springer, Cham, pp. 216–231. http://dx.doi.org/10.1007/978-3-319-05843-6_16.

Cox, K., Aurum, A., Jeffery, R., 2003. A use case description inspection experiment. In: Proceedings of the Australian Workshop on Software Requirements.

Cox, K., Phalp, K., 2000. Replicating the crews use case authoring guidelines experiment. Empir. Softw. Eng. 5 (3), 245–267. http://dx.doi.org/10.1023/A:1026542700033.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., Manning, C.D., 2014. Universal stanford dependencies: A cross-linguistic typology. In: Proceedings of the Ninth International Conference on Language Resources and Evaluation. LREC'14, pp. 4585–4592.

Denaro, G., Pezze, M., 2004. Petri nets and software engineering. In: Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Vol. 4, Springer, pp. 439–466.

Denger, C., Paech, B., Freimut, B., 2005. Achieving high quality of use-case-based requirements. Informatik - Forsch. Entwickl. 20 (1–2), 11–23. http://dx.doi.org/10.1007/s00450-005-0198-4.

Dingle, N.J., Knottenbelt, W.J., Suto, T., 2009. PIPE2: A tool for the performance evaluation of generalised stochastic Petri nets. ACM SIGMETRICS Perform. Eval. Rev. 36 (4), 34–39.

do Prado Leite, J.C.S., Doorn, J.H., Hadad, G.D., Kaplan, G.N., 2005. Scenario inspections. Requir. Eng. 10 (1), 1–21. http://dx.doi.org/10.1007/s00766-003-0186-9.

do Prado Leite, J.C.S., Hadad, G.D., Doorn, J.H., Kaplan, G.N., 2000. A scenario construction process. Requir. Eng. J. 5 (1), 38–61. http://dx.doi.org/10.1007/s00766-003-0186-9.

Domı, E., Pérez, B., Rubio, Á.L., et al., 2012. A systematic review of code generation proposals from state machine specifications. Inf. Softw. Technol. 54 (10), 1045–1066.

Femmer, H., Fernandez, D.M., Wagner, S., Eder, S., 2017. Rapid quality assurance with requirements smells. J. Syst. Softw. 123, 190–213. http://dx.doi.org/10.1016/j.jss.2016.02.047.

Firesmith, D., 2007. Common requirements problems, their negative consequences and the industry best practices to help solve them. J. Object Technol. 6 (1), 17–33.

Garousi, V., Bauer, S., Felderer, M., 2020. NLP-assisted software testing: A systematic mapping of the literature. Inf. Softw. Technol. 126, http://dx.doi.org/10.1016/j.infsof.2020.106321.

Glinz, M., 2000. Improving the quality of requirements with scenarios. In: Proceedings of the Second World Congress for Software Quality. 2WCSQ, Yokohama, pp. 55–60.

Gutiérrez, J.J., Nebut, C., Escalona, M.J., Mejías, M., Ramos, I.M., 2008. Visualization of use cases through automatically generated activity diagrams. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 83–96.

Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8 (3), 231–274. http://dx.doi.org/10.1016/0167-6423(87)90035-9.

He, X., 2013. A comprehensive survey of petri net modeling in software engineering. Int. J. Softw. Eng. Knowl. Eng. 23 (05), 589–625.

Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., Margaria, T., 2008. Software engineering and formal methods. Commun. ACM 51 (9), 54–59.

IEEE Computer Society, 1998. IEEE Recommended Practice for Software Requirements Specifications. IEEE.

Insfrán, E., Pastor, O., Wieringa, R., 2002. Requirements engineering-based conceptual modelling. Requir. Eng. 7, 61–72. http://dx.doi.org/10.1007/s007660200005.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., 1992. Object-Oriented Software Engineering: A Use-Case Driven Approach. Addison-Wesley.

Keller, R.M., 1976. Formal verification of parallel programs. Commun. ACM 19, 371–384. http://dx.doi.org/10.1145/360248.360251.

Kruchten, P., 2003. The Rational Unified Process: An Introduction, third ed. Pearson Education Inc., Boston, MA 02116.

Lami, G., Gnesi, S., Fabbrini, F., Fusani, M., Trentanni, G., 2004. An automatic tool for the analysis of nature languaje requirements. C.N.R. - Information Science and Technology Institute "A. Faedo".

Lee, W.J., Kwon, Y.R., Cha, S.D., 1998. Integration and analysis of use cases using modular Petri nets in requirements engineering. IEEE Trans. Softw. Eng. 24 (12), 1115–1130. http://dx.doi.org/10.1109/32.738342.

Lee, J., Pan, J.-I., Kuo, J.-Y., 2001. Verifying scenarios with time Petri-nets. Inf. Softw. Technol. 43 (13), 769–781.

Levenshtein, V.I., 1966. Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Doklady 10, 707–710.

Liu, S., Sun, J., Liu, Y., Zhang, Y., Wadhwa, B., Dong, J.S., Wang, X., 2014. Automatic early defects detection in use case documents. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ASE '14, ACM, New York, NY, USA, pp. 785–790. http://dx.doi.org/10.1145/2642937.2642969.

Lucassen, G., Dalpiaz, F., van der Werf, J.M.E.M., Brinkkemper, S., 2015. Forging high-quality user stories: Towards a discipline for agile requirements. In: 2015 IEEE 23rd International Requirements Engineering Conference. RE, pp. 126–135. http://dx.doi.org/10.1109/RE.2015.7320415.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D., 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, pp. 55–60. http://dx.doi.org/10.3115/v1/p14-5.

Marcus, M., Santorini, B., Marcinkiewicz, M.A., 1993. Building a large annotated Corpus of English: The Penn Treebank. Comput. Linguist. 19, 313–330.

Murata, T., 1989. Petri nets: Properties, analysis and applications. Proc. IEEE 77 (4), 541–580. http://dx.doi.org/10.1109/5.24143.

Phalp, K.T., Vincent, J., Cox, K., 2007. Assessing the quality of use case descriptions. Softw. Qual. J. 15 (1), 69–97. http://dx.doi.org/10.1007/s11219-006-9006-z.

PMI, 2014. Requirements management: Core competency for project and program success. URL https://www.pmi.org/learning/thought-leadership/pulse/core-competency-project-program-success, In-Depth Report. Internet.

Pohl, K., Rupp, C., 2015. Requirements Engineering Fundamentals. Rocky Nook Inc.

Portugal, R.L., Silva, L., Almentero, E., do Prado Leite, J.C.S., 2018. NFRfinder: A knowledge based strategy for mining non-functional requirements. In: XXXII Brazilian Symposium on Software Engineering. pp. 102–111.

Rago, A., Marcos, C., Diaz-Pace, J.A., 2016. Identifying duplicate functionality in textual use cases by aligning semantic actions. Softw. Syst. Model. 15, 579–603.

Rosadini, B., Ferrari, A., Gori, G., Fantechi, A., Gnesi, S., Trotta, I., Bacherini, S., 2017. Using NLP to detect requirements defects: An industrial experience in the railway domain. In: Proceedings of the 43rd Annual Southeast Regional Conference - Requirements Engineering: Foundation for Software Quality 2017, vol. 10153. Springer International Publishing, pp. 344–360. http://dx.doi.org/10.1007/978-3-319-54045-0_24.

Runeson, P., Höst, M., 2008. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14, 131–164.

Saavedra, R., Ballejos, L.C., Ale, M., 2013. Quality properties evaluation for software requirements specifications: An exploratory analysis. In: Anais do WER 2013 - Workshop em Engenharia de Requisitos. pp. 6–19.

Sarmiento, E., 2016. Analysis of Natural Language Scenarios (Ph.D. thesis). PUC-Rio.

Sarmiento, E., 2019. Construcción De Una Herramienta Para El Análisis De Requisitos De Software Descritos En Lenguaje Natural (Undergraduate thesis). UNSA.

Sarmiento, E., do Prado Leite, J.C.S., Almentero, E., 2015. Analysis of scenarios with Petri-Net models. In: 2015 29th Brazilian Symposium on Software Engineering. SBES, Belo Horizonte, pp. 90–99. http://dx.doi.org/10.1109/SBES.2015.13.

Sarmiento, E., do Prado Leite, J.C.S., Almentero, E., Alzamora, G.S., 2016. Test scenario generation from natural language requirements descriptions based on Petri-nets. Electron. Notes Theor. Comput. Sci. 329, 123–148. http://dx.doi.org/10.1016/j.entcs.2016.12.008.

Sarmiento-Calisaya, E., Cárdenas, E.H., Cornejo-Aparicio, V., Alzamora, G.S., 2020. Towards the improvement of natural language requirements descriptions: The C&L tool. In: SAC'20: Proceedings of the 35th Annual ACM Symposium on Applied Computer. pp. 1405–1413. http://dx.doi.org/10.1145/3341105.3374028.

Sinha, A., Sutton, S.M., Paradkar, A., 2010. Text2Test: Automated inspection of natural language use cases. In: Third International Conference on Software Testing Verification and Validation. ICST, pp. 155–164.

Sinning, D., Chalin, P., Khendek, F., 2009. LTS semantics for use case models. In: Proceedings of the 2009 ACM Symposium on Applied Computing. SAC '09, ACM, New York, NY, USA, pp. 365–370. http://dx.doi.org/10.1145/1529282.1529362.

Siqueira, F.L., Silva, P.S.M., 2011. An essential textual use case meta-model based on an analysis of existing proposals. In: Anais do Workshop em Engenharia de Requisitos. WER11.

Somé, S.S., 2010. Formalization of textual use cases based on Petri nets. Int. J. Softw. Eng. Knowl. Eng. 20, 695–737. http://dx.doi.org/10.1142/S0218194010004931.

The Standish Group, 2016. Chaos report. URL https://www.standishgroup.com/outline, Internet.

Tiwari, S., Arora, R., Bharambe, A., 2020. UC2Map: Automatic translation of use case maps from specification. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. pp. 1650–1653.

Tiwari, S., Gupta, A., 2015. A sistematic literature review of use case specification research. Inf. Softw. Technol. 67, 128–158. http://dx.doi.org/10.1016/j.infsof.2015.06.004.

Tiwari, S., Gupta, A., 2020. Use case specifications: How complete are they? J. Softw.: Evol. Process 32, http://dx.doi.org/10.1002/smr.2218.

Tjong, S.F., Berry, D.M., 2013. The design of SREE — A prototype potential ambiguity finder for requirements specifications and lessons learned. In: Requirements Engineering: Foundation for Software Quality. REFSQ 2013, Vol. 7830. Springer, Berlin, Heidelberg, pp. 80–95. http://dx.doi.org/10.1007/978-3-642-37422-7_6.

Van Galen, W., 2012. Use case goals. Scenarios and Flows.

Wang, C., Pastore, F., Goknil, A., Briand, L., 2020. Automatic generation of acceptance test cases from use case specifications: An NLP-based approach. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2020.2998503.

Weyns, D., Iftikhar, M.U., De La Iglesia, D.G., Ahmad, T., 2012. A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International Conference on Computer Science and Software Engineering. pp. 67–79.

Wiegers, K., Beatty, J., 2013. Software Requirements, third ed. Pearson Education.

Wilson, W.M., Rosernberg, L.H., Hyatt, L.E., 1997. Automated analysis of requirement specifications. In: Proceedings of the 19th International Conference on Software Engineering. ICSE '97, ACM, New York, NY, USA, http://dx.doi.org/10.1145/253228.253258.

Yue, T., Briand, L.C., Labiche, Y., 2010. An automated approach to transform use cases into activity diagrams. In: European Conference on Modelling Foundations and Applications. Springer, pp. 337–353.

Yue, T., Briand, L.C., Labiche, Y., 2013. Facilitating the transition from use case models to analysis models: Approach and experiments. ACM Trans. Softw. Eng. Methodol. (TOSEM) 22, 1–38. http://dx.doi.org/10.1145/2430536.2430539.

Zowghi, D., Gervasi, V., 2003. On the interplay between consistency, completeness, and correctness in requirements evolution. Inf. Softw. Technol. 45, 993–1009. http://dx.doi.org/10.1016/S0950-5849(03)00100-9.

Edgar Sarmiento-Calisaya: Assistant Professor at Universidad Nacional de San Agustín de Arequipa, Peru, Sarmiento-Calisaya received the master degree in Informatics from Federal University of Rio de Janeiro - UFRJ - Brazil in 2009 and the Ph.D. degree in Informatics from the PUC-Rio, Brazil in 2016. His research interests include Requirements Engineering, Software Testing, Model-driven Engineering and Context-Aware Software Systems.

Julio Cesar Sampaio do Prado Leite: Visiting Professor at Instituto de Computação, UFBA, Brazil, Leite received the 2017 IEEE Computer Society Life Time Award in Requirements Engineering. He has advised 26 Ph.Ds. and 34 Masters. Life Member of the IEEE Computer Society, the ACM and founding member of SBC (Brazilian Computer Society). He is an IFIP 2.9 group member since its foundation and is an editorial member of the Requirements Engineering Journal. He is also one of the founding members of the WER (Workshop on Requirements Engineering). His main area of research is software transparency, researching ways to improve the quality of software products by making them more transparent to society at large.