PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Edgar Sarmiento Calisaya

Analysis of Natural Language Scenarios

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Doutor.

Advisor: Prof. Julio Cesar Sampaio do Prado Leite

Rio de Janeiro

April 2016

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Edgar Sarmiento Calisaya

Analysis of Natural Language Scenarios

Thesis presented to the Programa de Pós-Graduação em Informática, of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Doutor.

Prof. Julio Cesar Sampaio do Prado Leite
Advisor
Departamento de Informática – PUC-Rio

Prof. Arndt von Staa
Departamento de Informática – PUC-Rio

Prof. Carlos José Pereira de Lucena
Departamento de Informática – PUC-Rio

Prof. Marcos Roberto da Silva Borges
UFRJ

Prof. Eduardo Kinder Almentero
UFRRJ

Prof. Vera Maria B. Werneck
UERJ

Prof. Márcio da Silveira Carvalho
Coordinator of the Centro Técnico Científico da PUC-Rio

Rio de Janeiro, April 13th, 2016.

Edgar Sarmiento Calisaya

Graduated in Systems Engineering (Computer Science) from Universidad Nacional de San Agustin – Arequipa – Perú. He obtained the degree of Master in Informatics at Universidade federal do Rio de Janeiro – Rio de janeiro – Brazil. He has been working in the field of Software Engineering for over fifteen years.

Ficha Catalográfica

Sarmiento Calisaya, Edgar

Analysis of Natural Language Scenarios / Edgar Sarmiento Calisaya ; advisor: Julio Cesar Sampaio do Prado Leite. – 2016.

231 f. : il. ; 30 cm

Tese (doutorado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2016.

Inclui bibliografia

1. Informática – Teses. 2. Requisitos de software. 3. Cenários. 4. Petri-Net. 5. Análise de requisitos. 6. Verificação de requisitos. I. Leite, Julio Cesar Sampaio do Prado. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents, Antonia Calisaya Sarmiento and Fabio Estanislao Sarmiento Choque.

Acknowledgements

First of all, I would like to give my most sincere tribute and gratitude to my advisor Julio Cesar Sampaio do Prado Leite who believed in this work. Their academic vision and timely discussions always inspire me. For his friendship, guidance, encouragement and insights, which guide me through my Ph.D life.

Futhermore, I would like to thank my parents Antonia and Fabio Estanislao for education and caring, for giving me all the necessary support, so that I could come here and be able to do this Ph.D study.

I would like to thank my brothers Lourdes, Juan Estanislao and Alberto for their companionship, love and patience in listening to me every time I needed.

I would like to thank to the professors Vera Maria B. Werneck, Arndt von Staa, Noemi Rodrigues and Marcos Roberto da Silva Borges for their important contributions; and who participated in the examination Committee.

To my mates Roxana, Giovana, Elizabeth, Joanna, Priscilla, Marilia, Andre, Eduardo and Henrique of Requirements Engineering research group at PUC-Rio, for the inspiration, knowledge, contributions, fellowship and time spend on the problem presented in this thesis.

Additionally, I am grateful to my friends Guina, Ruben Rafael, Giovana, Roxana, Fernanda and Gilbert, who were always very helpful and supportive.

I am also thankful to professors at PUC-Rio for everything I learned from them.

In adittion, I would like to thank to CAPES funding agency and the PUC-Rio, for the financial support.

Abstract

Sarmiento Calisaya, Edgar; Leite, Julio Cesar Sampaio do Prado. **Analysis of Natural Language Scenarios**. Rio de Janeiro, 2016, 231p. DSc Thesis - Departamento de informática, Pontifícia Universidade Católica do Rio de Janeiro.

Requirements analysis plays a key role in the software development process. Natural language-based scenario representations are often used for writing software requirements specifications (SRS). Scenarios written using natural language may be ambiguous, and, sometimes, inaccurate. This problem is partially due to the fact that relationships among scenarios are rarely represented explicitly. As scenarios are used as input to subsequent activities of the software development process (SD), it is very important to enable their analysis; especially to detect defects due to wrong information or missing information. This work proposes a Petri-Net and Natural Language Processing (NLP) based approach as an effective way to analyze the acquired scenarios, which takes textual description of scenarios (conform to a metamodel defined in this work) as input and generates an analysis report as output. To enable the automated analysis, scenarios are translated into equivalent Place/Transition Petri-Nets. Scenarios and their resulting Petri-Nets can be automatically analyzed to evaluate some properties related to *unambiguity*, *completeness*, *consistency* and *correctness*. The identified defects can be traced back to the scenarios, allowing their revision. We also discuss how *unambiguity*, *completeness*, *consistency* and *correctness* of scenario-based SRSs can be decomposed in related properties, and define heuristics for searching defect indicators that hurt these properties. We evaluate our work by applying our analysis approach to four case studies. The evaluation compares the results achieved by our tool-supported approach, with an inspection based approach and with related work.

Keywords

Software requirements; scenarios; use cases; requirements analysis; requirements verification; Petri-Net; Natural Language Processing.

Resumo

Sarmiento Calisaya, Edgar; Leite, Julio Cesar Sampaio do Prado. **Análise de Cenários em Linguagem Natural**. Rio de Janeiro, 2016, 231p. Tese de Doutorado - Departamento de informática, Pontifícia Universidade Católica do Rio de Janeiro.

A análise de requisitos desempenha um papel fundamental no processo de desenvolvimento de software. Neste sentido, representações de cenários baseados em linguagem natural são muitas vezes utilizados para descrever especificações de requisitos de software (SRS). Cenários descritos usando linguagem natural podem ser ambíguos e, às vezes, imprecisos. Este problema é parcialmente devido ao fato de que os relacionamentos entre os cenários são raramente representados explicitamente. Como os cenários são utilizados como entrada para as actividades subsequentes do processo de desenvolvimento de software (SD), é muito importante facilitar a sua análise; especialmente para detectar defeitos devido a informações erradas ou falta de informação. Este trabalho propõe uma abordagem baseada em Redes de Petri e técnicas de Processamento de Linguagem Natural como uma forma eficaz para analisar os cenários adquiridos, e que toma descrições textuais de cenários (em conformidade com um metamodelo definido neste trabalho) como entrada e gera um relatório de análise como saída. Para facilitar a análise automática, os cenários são transformados em Redes de Petri (Lugar/Transição) equivalentes. Os cenários e suas Redes de Petri resultantes podem ser analisados automaticamente para avaliar algumas propriedades relacionadas à *desambiguidade*, *completeza*, *consistência* e *corretude*. Os defeitos identificados podem ser rastreados até os cenários, permitindo a sua revisão. Nós também discutimos como *desambiguidade*, *completeza*, *consistência* e *corretude* das SRSs baseadas em cenários podem ser decompostas em propriedades relacionadas, e definimos heurísticas para encontrar indicadores de defeitos que prejudicam estas propriedades. Avaliamos nosso trabalho, aplicando a nossa abordagem de análise em quatro estudos de caso. Essa avaliação compara os resultados obtidos pela nossa abordagem automatizada contra os resultados obtidos por um processo de inspeção e com trabalhos relacionados.

Palavras-chave

Requisitos de software; cenários; casos de uso; análise de requisitos; verificação de requisitos; Petri-Net; Processamento de Linguagem Natural.

Contents

List of Figures

List of Tables

List of Abbreviations

BNF	Backus Normal Form
C&L	Cenários e Léxico
CSP	Communicating Sequential Processes
DEO	Discrepancies, Errors, and Omissions
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organization
FR	Functional requirements
GORE	Goal-Oriented Requirements Engineering Approaches
LTS	Labeled Transition Systems
LEL	Language Extended Lexicon
LSC	Live Sequence Charts
MSC	Message Sequence Charts
MVC	Model-View-Controller
NL	Natural Language
NLP	Natural Language Processing
NFR	Non-functional requirements
POS	Part-of-Speech - POS
PN	Petri-Nets
PNML	Petri Net Markup Language
RE	Requirements Engineering
RNL	Restricted-form of natural language
SADT	Structured Analysis and Design Technique
SIG	Soft-goals Interdependency Graph
SD	Software Development
SRS	Software Requirements Specification
UML	Unified Modeling Language
UofD	Universe of Discourse

1
Introduction

Many research studies have shown how the *Requirements Engineering* (RE) activities play an important role in the reliability, cost and safety of a software system; especially, the importance of early ***requirements analysis*** on the reduction of the development costs, confusion and complexity in the later activities of *Software Development* (SD). RE activities are important mainly for two reasons. *First*, they help software development stakeholders to better understand and communicate the software requirements. *Second*, their main output, i.e. the *Software Requirements Specification* (SRS) serves as the basis for later software development activities, e.g., design, coding and testing.

Therefore, it is highly desirable to produce a Software Requirements Specification with a good quality, i.e., a SRS that is more correct, consistent, complete, unambiguous, understandable and traceable (IEEE, 1998, Lee et al., 1998; Glinz, 2000; Lee et al., 2001; Cheung et al, 2006; Somé, 2010; Zhao and Duan, 2009).

Requirements Analysis includes activities related to verification and validation (Leite, 2007), such as *finding defects* in structural and behavioral properties of SRS documents and *addressing problems* related to these properties, which could reduce most of the risks in the later activities of software development. However, *requirements analysis* is still an extensive and iterative process, which is mostly performed manually, requiring a great effort and taking a lot of time.

Requirements described through rigorous or tabular specifications enable automated analysis by simulating a sequence of events that represents a narrow aspect of a system's required behavior; these detect several classes of faults by checking specification properties (Lamsweerde et al., 1998; Heitmeyer, 2007). However, these practices are expensive and not widely used in industrial practice.

For practical reasons, and in order to allow for an easy communication with stakeholders, informal or semiformal representations are widely used by *user-*

oriented approaches. User-oriented approaches are dominant during Requirements Engineering activities in industry; and, one of the key elements in this perspective is the notion of *scenarios*. In this context, a SRS is represented as a collection of scenarios and described by specific flows of events and their guard conditions. The use of scenarios helps understanding a specific situation in an application, prioritizing their behavior (Leite et al., 2000). Some of the most prominent languages to write scenarios are restricted-form of use case descriptions (Cockburn, 2001), scenario descriptions (Leite et al., 2000), UML dynamic behavior (UML, 2015) diagrams and Message Sequence Charts (Andersson and Bergstrand, 1995).

The graphical notation based languages for writing scenarios are very attractive and user-friendly; however, they can be difficult to design, and domain experts cannot reasonably be asked to draw them (Gutiérrez, 2008). Although these languages provide an accessible visualization of models, they lack formal semantics to support the analysis of structural and behavioral properties of the modeled system.

According to Glinz, (2000), in the literature, there is no clear distinction between natural language-based scenarios and use case descriptions, both scenarios and use cases describe situations that could possibly happen between the users and a system. However, the scenario language proposed by Leite et al (2000) also helps on understanding specific situations in an application, prioritizing their behavior.

In this thesis, **scenario** and **use case** are considered **synonymous** because they are described by similar components. There are several different templates or syntax for writing scenarios, and some of the most common components used to detail scenarios are: ***Title****/Name,* ***Goal, Pre-condition, Post-condition, Actors, Episodes****/Main Flow* and ***Exceptions****/Alternative Flows*.

Unfortunately, natural language-based scenarios exhibit some shortcomings: (1) informally specified scenarios are usually hard to analyze, because natural language is by definition ambiguous; (2) modularity is poorly supported, because the relationships among scenarios are rarely represented explicitly; and (3) currently**,** there are no systematic approaches to identify and make explicit potential concurrency issues (e.g. deadlock, non-determinism) in initial requirements descriptions.

According to Lee et al. (1998), although such subsets of scenarios might *seemingly* be independent, they are rarely truly independent in practice. A set of scenarios can be considered as a set of ***concurrently*** executing threads. Thus, from the concurrency perspective, scenarios are rarely truly independent in practice; they may interact or compete with each other by *communication channels* or *shared resources*, what can lead to erroneous situations such as *deadlocks*.

1.1. Motivation

Because of inherent ***ambiguity*** of natural language (NL), defects are inevitably introduced into scenario-based SRS. Thus, assessing the quality of a SRS document is not a simple process, mainly, because:

- Finding defects in scenarios is an important activity mostly performed manually, which is expensive, time-consuming and error-prone.
- Multiple users with different viewpoints and conflicting needs about the system are involved at RE activities.
- Relationships among scenarios are rarely represented explicitly (Lee et al., 1998, Leite et al., 2000). Scenarios are related to other scenarios by sequential (precedence order) and non-sequential relationships (indistinct sequential order, concurrency or parallelism).
- Heuristics for finding non-explicit relationships among scenarios are rarely proposed (Leite et al., 2005).
- Finding defects from the relationships among different scenarios is a complex activity. It is necessary to execute (or simulate the behavior) a set of scenarios for detecting defects from the relationships among them (Denger et al., 2005).
- Most of the defects found by analysis techniques are in fact simple linguistic defects in single scenarios (Adapted from Gnesi et al., 2005).

Ambiguity may lead to *incomplete*, *inconsistent* and *incorrect* scenario-based SRS documents. Since a scenario-based SRS describes requirements statements using scenarios and their relationships: ***Ambiguity*** occurs when two or more users have different interpretations of the same requirement statement stated in a single scenario. ***Incompleteness*** in a single scenario or involving multiple

scenarios occurs because the world is complex; as such, users or clients are not able to identify and develop all relevant requirements within scenarios. ***Inconsistency*** occurs when two or more users have conflicting or overriding requirements, thus, scenarios can overlap other scenarios. ***Incorrect*** scenarios occur when the acquired requirements do not accurately reflect the facts, or erroneous predicts about future states.

Some examples of defects that hurt ***Unambiguity***, ***Completeness***, and ***Consistency*** quality properties in scenario-based SRS include:

- **Unambiguity**: Different interpretations of the same requirement;
    - ***Title*** contains *subjective* words or phrases (e.g., similar, better);
    - ***Episode*** contains *weak* words or phrases (e.g., can, might);
- **Completeness**: Fully developed requirement statements;
    - ***Actor*** *does not participate* in any ***episode*** of the main flow;
    - ***Conditional episode is not conform*** to the syntax rules;
    - ***Related Scenario*** does not exist in the set of ***Scenarios***;
- **Consistency**: Free of conflicting or overriding requirement statements;
    - ***Pre-condition*** *coincidence*: non-determinism (warning);
    - ***Bi-directional reference*** among related scenarios (circular inclusion);
    - ***Never enabled*** sequence of episodes (or steps);

Usually, these defects are fixed in software design activities; however, *ambiguity*, *incompleteness*, *inconsistency* and *incorrectness* in scenarios must be resolved in early activities of software development (i.e., RE activities). It increases the software reliability and improves the productivity of software development (Lee et al., 1998). The importance of SRS quality has been recognized by several studies (Boehm and Basili, 2001; Bernstein and Yuhas, 2005).

1.2. Problem

Scenario specifications are usually informal or semi-formal, and in these cases, they are not the best choice for further automated analysis (including graphical notation based models) because they lack of formal semantics to support

it. Thus, there is a lack of formal semantics to support the early analysis of structural and behavioral properties of systems described as scenarios.

Several research studies have shown the importance to formalize scenarios through restricted-form of use case descriptions (Somé, 2010), *Message Sequence Charts - MSC* (Andersson, and Bergstrand, 1995; Damas et al., 2006), Live Sequence Charts - LSC (Damm and Harel, 2001) or BNF-like gramar (Hsia et al., 1994); other researchers have used concepts from *Petri-Nets* (Lee et al., 1998; Lee et al., 2001; Cheung et al., 2006; Zhao and Duan, 2009; Somé, 2010), *Statecharts* (Glinz, 2000; Denger et al., 2005), Labeled Transition Systems - LTS (Sinnig et al., 2007) or Communicating Sequential Processes - CSP (Cabral and Sampaio, 2006).

These literature circles argue for the need for a precise representation for scenarios in order to be useful in *automated analysis*, model derivation or test generation. In these approaches, scenarios are described by a variety of scenario notations, in some cases with rigorous semantics; scenarios are used to document system requirements, then, scenarios are translated into *Petri-Nets - PN* (Murata, 1989), *LTS* (Keller, 1976) or *CSP* (Roscoe, 1998); which are used as the mechanism to enable rigorous analysis. The resulting formal models can be further processed and analyzed using available tools to verify structural and behavioral properties, ensuring mainly the *consistency* and *correctness*.

The translation-based approaches are difficult to apply because it requires a strong knowledge and experience on formal modeling for translating initial scenarios into formal models. Other drawbacks are:

- There is no consensus on how to represent scenarios; some languages depend on formal definition of pre-conditions and post-conditions within single scenarios (Lee et al., 1998; Sinnig et al., 2009);
- Most of the existing approaches do not provide formal definition of translation rules between scenario elements and formal model elements, which can make the automation more difficult;
- Most of the existing approaches do not provide procedures for integrating a set of related scenarios into a whole representation, and detect defects from these relationships. Scenarios interact by sequential and non-sequential relationships;

- In most of the existing approaches, the use of the analysis feedback of equivalent formal models to improve the scenario descriptions is difficult, since they do not provide ways of tracing to defects in the original scenario.

1.3. Objective

Motivated by the importance of improving the quality of Software Requirements Specification documents based on scenario representations, we are propose **a new approach for scenarios analysis that is based on Petri-Nets *and Natural Language Processing (NLP) techniques, which* exploits inter-scenario relationships to overcome major unsolved problems and improve the state of the art**.

1.4. Thesis

"*NATURAL LANGUAGE-BASED SCENARIOS CAN BE ANALYZED THROUGH PETRI-NETS AND NLP" BY AN APPROACH THAT*:

- Show defects that hurt **unambiguity** in single scenarios at RE;
- Show defects that hurt **completeness** in single scenarios at RE;
- Show defects that hurt **completeness** from relationships among scenarios at RE;
- Show defects that hurt **consistency** and **correctness** in Petri-Nets derived from scenarios and their relationships at RE;
- Support **modularity** by proposing *heuristics for finding explicit and non-explicit relationships* among scenarios;
- Support **traceability** by indicating defects in Petri-Nets and showing the source of the defects in scenarios;
- Can be implemented through a software tool.

1.5. Proposed Solution

As scenarios are useful in other development activities, these scenarios must be correct and valid. Therefore, effectiveness of scenarios analysis could be significantly improved by an approach, which could discover defects that are hidden in scenarios and their relationships in an automatic way. The higher goal

of this thesis is to develop a "**Petri-Net and NLP based Approach as an Effective Way to Analyze the Acquired Scenarios**", which evaluates structural and behavioral properties related to ***Unambiguity***, ***Completeness***, ***Consistency*** and ***Correctness***. The following goals refine the stated goal:

- Define a restricted-form of natural language (RNL) to write scenarios;
- Develop heuristics for finding non-explicit relationships among scenarios;
- Develop a systematic procedure that transforms scenarios stated in a RNL to Petri-Nets;
- Improve the existing NLP Parsing strategies to correctly identify the Subject, Objects and Action-Verb in textual scenario sentences.
- Employ the non-functional requirements (NFR) approach to:
    - Model the relationships between unambiguity, completeness, consistency and correctness qualities of scenarios;
    - Identify the properties related to unambiguity, completeness, consistency and correctness;
- Develop heuristics for searching *defect indicators* that hurt properties related to *unambiguity*, *completeness*, *consistency* and *correctness* qualities.

1.5.1. Approach Overview

Our scenarios analysis approach checks the acquired *scenarios* by detecting wrong information, missing information and erroneous situations that can be hidden within scenarios and their relationships. In this regard, we instantiate a **Quality Model for Scenarios** (defined in this thesis), and consider the results achieved by *NLP* and *Petri-Net* based related work.

The related work in using the potential of Petri-Nets for scenario formalization indicates that Petri-Nets are an effective mechanism for scenario-based SRS analysis. The motivation behind translating scenarios into Petri-Nets can be attributed to three reasons: ***First***, the reachability analysis can reveal the incorrect behavior of a set of scenarios (mapped into Petri-Nets); ***Second***, the availability of Petri-Net tools, such as PIPE2 (2015); and ***Third***, the portability of Petri-Net models (interchangeable format between tools - *Petri Net Markup Language - PNML*).

So, in our approach: ***First***, requirements engineers start to describe the different functionalities, services or situations of the system as *scenarios* using a RNL. ***Second***, irrelevant information within scenario elements are removed**.** ***Third***, in order to perform an automated analysis of scenarios, an initial system design is derived by translating these *scenarios* into *Place/Transition Petri-Nets*, and synthesizing them into a consistent whole *Petri-Net*. ***Fourth***, *scenarios* and their resulting *Petri-Nets* are automatically analyzed to evaluate some properties related to *unambiguity*, *completeness*, *consistency* and *correctness*. ***Fifth***, the analysis outcome is formatted and returned to the requirements engineers. ***Sixth***, if defects are found, the analysis feedback is used to improve the scenario descriptions, since the identified defects and their causes can be traced to the scenarios. Figure 1 depicts an overview of our approach. The different phases of our approach were implemented in the C&L (2010) prototype tool.

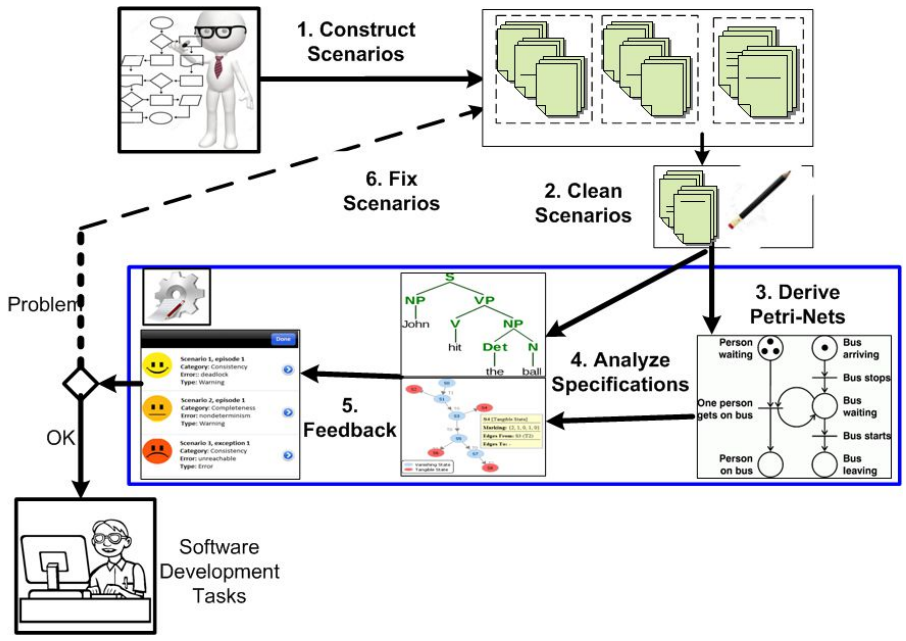


Figure 1 – Overview of the Scenarios Analysis Approach.

1.5.2. Expected Contribution

The main contribution of this thesis is an automated analysis approach of structural and behavioral properties in scenario specifications. The analysis is able to detect defects that provide evidence that the properties related to ***Unambiguity***, ***Completeness***, ***Consistency*** and ***Correctness*** were violated.

This approach benefits from both the precision of graphical Petri-Nets and the usability of textual scenarios; and it also allows an easier integration to available Petri-Net tools like PIPE2 (2015).

The objective of this thesis is to contribute with the following results:

- The definition of a *Restricted-form of Natural Language-based Scenario* Model, whose elements may be written using a semi-structured linguistic grammar. This scenario language was initially proposed by Leite et al (2000).
- The definition of *Heuristics for Finding Non-explicit Relationships* among scenarios: Scenarios are related to other scenarios by sequential (precedence order) and non-sequential relationships. Frequently, non-sequential relationships are non-explicit.
- The definition of a reusable *Quality Model for Scenarios*, which organizes the properties related to *Unambiguity*, *Completeness*, *Consistency* and *Correctness*. These properties were based on previous work (Leite et al., 2000) and related work.
- A procedure to *Translate Scenarios* stated in a restricted-form of natural language into Petri-Nets, preserving the consistency between these equivalent representations.
- A systematic procedure to synthesize a system design from the resulting Petri-Nets, preserving the original properties of synthesized Petri-Nets.
- Manage the *State Explosion Issue* (Lee et al., 1998): State explosion issue is a serious problem when applying *Petri-Net analysis* to large systems. A contribution of this thesis is a MULTI-STEP BOTTOM-UP analysis approach to manage this problem.
- The development of the C&L tool (Cenários & Léxicos): An experimental tool that automatically detects potential defects in scenarios. For every potential defects detected, C&L shows it to the user in a understandable way.

1.5.3. Evaluation

Five case studies with different degree of complexity were carried out to evaluate the *accuracy* and the *scalability* of the proposed analysis approach. We

evaluated the *accuracy* of results produced by the developed tool (C&L-Lua) with respect to reference solutions elaborated by expert Requirements Engineers of different universities. These set of scenarios have a near-typical profile, i.e., they contain typical defects in industrial projects (UCDB, 2015).

1.6. Outline

This thesis is organized as follows:

Chapter 2: Presents a general introduction of Requirements Engineering, Concurrency and Petri-Nets. It also presents the state of the art in requirements analysis, compare the different existing approaches, and identify the most important research gaps.

Chapter 3: Presents a reusable Quality Model for Scenarios by modeling the relationships between unambiguity, completeness, consistency and correctness qualities.

Chapter 4: Presents the proposed approach for scenarios analysis. It includes the proposed scenario language, which is used to write scenarios enabling further transformations into executable representations; the procedure for translating scenarios into Petri-Nets; the strategy for evaluating structural and behavioral properties of scenarios; and the strategy for managing the state explosion issue of Petri-Nets.

Chapter 5: Presents the C&L tool architecture and its implementation. An experimental tool that automatically detects potential defects in scenarios

Chapter 6: Shows the evaluation of the developed approach by finding defects in scenario specifications.

Chapter 7: Discusses the differences between our work and those related work and summarize our improvements on the state-of-the-art. Presents the conclusions, limitations, and some suggestions for future work.

2
Theoretical Background

This chapter begins with a general overview of Requirements Engineering and techniques to document and analyze scenarios (Section 2.1). Next, Section 2.2 investigates the topic of requirements quality. Concurrency and Petri-Nets are introduced in a comprehensive way in Section 2.3. Section 2.4 discusses research (or tools) related to the analysis of Natural Language based SRSs, compares techniques to analyze static and dynamic aspects related to quality attributes of requirements in a comprehensive way, and highlight research gaps. Finally, Section 2.5 concludes by discussing the relationships between scenarios and concurrent systems.

2.1.
Requirements Engineering

The development of software systems with acceptable quality and lower cost is a constant concern for the software development industry as well as it customers. An erroneous or incomplete understanding of the problem that software aims to solve may lead to software systems correctly implemented, but missing the customer needs. One of the main success factors at the activities involved in the ***SD*** process, is the correct understanding of the problem domain, i.e., a more clear and precise ***SRS***; but this is not always possible because of multiple stakeholders involved in the software development process, with different needs, assumptions and points of view of the domain. Thus, Requirements engineering is closely related to the good quality of software systems; thus, it is a key factor for successful software development companies.

Requirements Engineering (RE) is one of the most crucial and complex activities in software development and bridges the gap between customers needs and software engineering. According to Pohl (1994), requirements engineering may be understood as a process with a set of activities; where the desired output of this process is a document (SRS - Software Requirements Specification) expressed using a formal language on which most of stakeholders agree. This

process inputs are user's points of view (usually ambiguous) of the system to be built; these inputs are obtained using an informal language (or making use of graphics), usually natural language.

Similarly, Leite (2007) subdivided the Requirements engineering into elicitation, modeling, analysis, and management activities (**Figure 2**). Elicitation is the first step and responsible to identify most of relevant key stakeholders and discover what they need. Modeling is the process of building abstract descriptions of the requirements that are amenable to interpretation. Analysis corresponds to the generation of a SRS with acceptable quality and it is subdivided into verification and validation. Requirements management is transversal to RE process, and it consists of version control, change control, and traceability of the requirements (Leite, 2007). The input of this process is the Universe of Discourse (UofD), i.e., it includes all the sources of information and all the people related to the software.

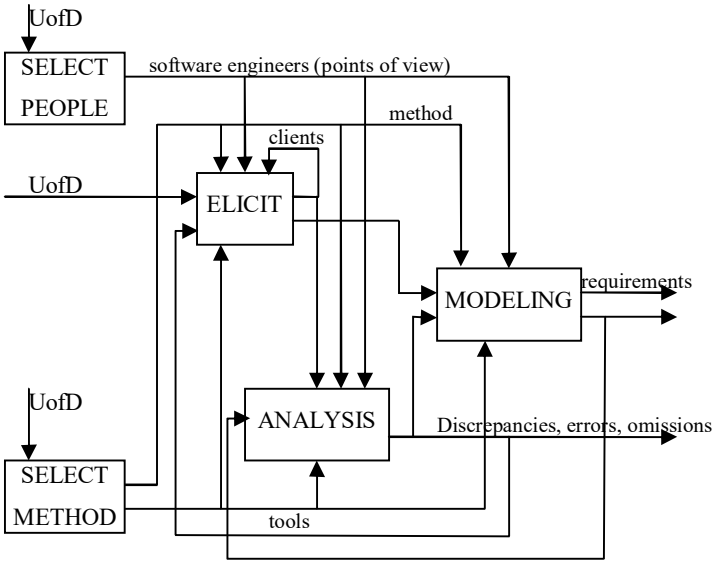


Figure 2 - Requirement Engineering (Leite, 2007)

2.1.1. Requirements

Poorly defined requirements are the major cause for software projects to fail. Requirements in software engineering are descriptions of actions, behavior and constraints of a system in order to meet stakeholders' needs. According to Sommerville (2010), requirement is the specification of what is to be implemented, and he classified requirements into functional and non-functional

requirements and distinguished between two different levels of abstraction: user requirements and system requirements.

- *Functional requirements* (FR) define the behavior of the system and what it should do.
- *Non-functional requirements* (NFR) also known as quality attributes (QA) or soft-goals (van Lamsweerde, 2001), describe the system attributes and define the constraints of a system.
- *User requirements* define the software functionality from a user perspective. They define what the software has to do to accomplish the user's goals.
- *System requirements* are more detailed description of software functions, services and operational constraints.

2.1.2. Requirements Specification

Once the requirements are gathered, they need to be described. Requirements are expressed in a software requirements document, which is also called a ***SRS***. The document includes requirements definitions, requirements specifications or system models. Usually, requirements definitions and requirements specifications are presented separately. However, in some cases these two are incorporated into a single description.

A ***requirements definition*** is a high-level abstract statement of the services a system must provide and the constraints it must meet. It is expressed in a natural language and without any reference to a solution, and targeted mainly at clients and project managers. The definition is based on information supplied by clients or users.

A ***requirements specification*** (also called a functional specification) is a structured description describing the services the system must meet in a more detailed manner. It may contain references to technologies or solutions. The description may serve as a contract between software developers and customers. The requirements specification expands the requirements definition and is targeted mainly at project managers and software developers.

Sommerville (2010) called requirements definition, a *user requirements specification*; and requirements specification, a *system requirements specification*.

To document software requirements there are various techniques and languages, which may be classified as being *informal*, *formal* or *semi-formal*.

- ***Informal*** techniques use unrestricted natural language to document requirements. The advantage of natural language is that it is universal and flexible, but unfortunately is ambiguous.
- ***Formal*** languages are based on rigorous mathematical or logical reasoning for which the syntax, semantics and rules are explicitly defined (e.g. Temporal Logic, First Order Logic, SCR, Z, B, Petri-Nets). These methods are accurate and eliminate ambiguity, but they are hard to understand without a specific training and require a formal background.
- ***Semi-formal*** techniques include diagrams and tabular techniques that present information in a structured form (e.g. UML diagrams, Message Sequence Charts diagrams, ER models). They try to close the gap between two previous techniques.

2.1.3. Scenario-Based Requirements Specification

While requirements are statements describing the expected system services, scenario-based representations have attracted considerable attention in **RE**. A scenario describes a situation that could possibly happen in a system.

In this context, a SRS document could be described as a collection of scenarios and each scenario is described by specific flows of events and their guard conditions. The use of scenarios helps understanding a specific situation in an application, prioritizing their behavior (Leite et al., 2000).

The main purpose of scenarios is to stimulate thinking about possible events, opportunities and risks in a system. They are often applied to model and communicate requirements among stakeholders due to their comprehensibility (Glinz, 2000; Leite et al., 2000).

2.1.3.1. Scenarios

According to Leite et al. (2000), the word *scenario* has a particular meaning in the software engineering community. It is "a description technique that is both process-focused and user centric". It is widely used in requirements engineering

because it helps engineers to better understand the software requirements and its interface with the environment.

According to Glinz (2000), in software engineering, scenarios are described as "an ordered set of interactions between partners, usually between a system and a set of actors external to the system". Other researchers have similar definitions. For instance, Leite at al. (2000) defines a scenario as a partial description of the application behavior that occurs at a given moment in a specific geographical context - a *situation*. Van Lamsweerde and Willemet (1998) framed the term to "temporal sequence of interactions among different agents in the restricted context of achieving some implicit purpose".

In literature there is no clear distinction between scenarios and use cases. While some authors consider that each scenario corresponds to one use case (Glinz, 2000), others define a scenario as sequences of use case steps that represent different paths through a use case (Cockburn, 2001). According to Glinz (2000), a scenario may comprise a concrete sequence of interaction steps (instance scenario) or a set of possible interaction steps (type scenario).

2.1.3.2. Representing Scenarios

There are a wide variety of scenarios representations in the literature, each one with quite different purposes. Therefore, scenarios can take many forms and provide various types of information on different levels of abstraction and formalism. Some of the most prominent languages to write scenarios are semi-structured-form of use case descriptions (Cockburn, 2001), restricted-form of scenario descriptions (Leite at al., 2000), UML sequence diagrams (UML, 2015), UML activity diagrams (UML, 2015), MSC (Andersson and Bergstrand, 1995), LSC (Damm and Harel, 2001), StateCharts (Harel, 1987) or Petri-Nets (Murata, 1989).

There are several styles in which scenarios are written. In system development, Alexander and Maiden (2005) defined six common types of scenarios used. These can have different representation style and are defined as follow:

- ***Story***: Narrative description of connected sequence of events, e.g. a user story that is written in plain text as often seen in agile methodologies.

- ***Sequence***: straight-line of interactive steps taken by human or system agents, e.g. List of numbered user actions.
- ***Structure***: More elaborated representation of a scenario, e.g. activity diagram.
- ***Situation***: Snapshot of a future state of the system, e.g. a picture or an example of a user interface of an imagined future state.
- ***Simulation***: Models to explore and animate stories or situations, e.g. animated diagram to show the eventual real effects of alternative conditions and courses of action.
- ***Storyboard***: Drawing or a sequence of drawings to describe a story, e.g. mock-ups of a flow that are linked together.

Natural language-based scenarios like use case (Cockburn, 2001) or scenario (Leite at al., 2000) representations, are widely used to specify software requirements because they promote the communication between engineers and stakeholders, even when they have no modeling background. Furthermore, natural language-based scenarios offer several practical advantages: (1) Scenarios are easy to describe and understand; (2) They are scalable; the behavior of a large and complex system can be represented as a collection of independently and incrementally developed scenarios; and (3) It is relatively easy to provide traceability throughout the design (Lee et al., 1998).

2.1.3.3. Use Case Representation

A typical use case (Cockburn, 2001) describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment. Every *use case* constitutes a goal-oriented set of interactions between external actors and the system under consideration. The term *actor* is used to describe any person or system that has a *goal* in the system under discussion or interacts with the system to achieve some other actor's goal. A primary actor triggers the system behavior in order to achieve a certain goal. A secondary actor interacts with the system but does not trigger the use case.

A use case is completed successfully when the goal that is associated with it is reached. Use case descriptions also include possible *extensions* to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that

may lead to failure in completing the service in case of exceptional behavior, or some fault. In the textual notation proposed by Cockburn (2001), the main flow is expressed, in the ‘‘description’’ section, by an indexed sequence of NL sentences, describing a sequence of actions of the system. Variations are expressed (in the ‘‘extensions’’ section) as alternatives to the main flow, linked by their index to the point of the main flow from which they branch as a variation (See **Table 1**).

Table 1 – Use Case Template (Cockburn, 2001)

Element	Description	
Use Case #	<The name is the goal as a short active verb phrase>	
Goal in Context	<A longer statement of the goal in context if needed>	
Scope & Level	<What system is being considered black box under design> <One of: Summary, Primary task, Sub-function>	
Preconditions	<What we expect is already the state of the world>	
Success End Condition	<The state of the world upon successful completion>	
Failed End Condition	<The state of the world if goal abandoned>	
Primary, Secondary Actors	<A role name or description for the primary actor> <Other systems relied upon to accomplish the use case >	
Trigger	<The action upon the system that starts the use case>	
Description	Step	Action
	1	<Put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
Extensions	Step	Branching Action
	1a	<Condition causing branching> <Action or name of sub-use case>
Sub-Variations	Step	Branching Action
	1	<List of variations>

2.1.3.4. Scenario Representation

The scenario language proposed by Leite et al. (2000) describes situations in the system and its relation with other situations and the environment. This description is made using natural language. The proposed structure of this model is composed of the following elements: *title*, *goal*, *context*, *resources*, *actors*, *episodes*, *exceptions* and *constraints* (See Table 2). In the *episodes*, the operational behavior of the situations is described in natural language, but using special operators for optionality, concurrency and selection. A scenario is identified by a *title* and must satisfy a *goal*. The path to achieving this goal must be described in detail in its *episodes*. The episodes represent the main stream of the actions, but also include variations and possible alternatives. An *exception* can occur during the execution of episodes, which indicates that there is an obstacle to satisfy the goal. The treatment to this exception does not need to satisfy the scenario goal.

Table 2- Scenario template (Leite et al., 2000)

Element	Description	
Title	<Identifies the scenario>	
Goal	<Describe the purpose of the scenario>	
Context	<Describes the scenario initial state> <Must be described through at least one of these options: precondition, geographical or temporal location>	
Resources	<Passive entities used by the scenario to achieve its goal>	
Actors	<Active entities directly involved with the situation>	
Episodes	<Sequential sentences in chronological order with the participation of actors and use of resources> <One of: Simple, Conditional, Optional> <Non-sequential order can be bounded by the symbol "#", it is used to describe parallel or concurrent episodes>	
Exception	Cause	Solution
	<Situations that prevent the proper course of the scenario>	<Its treatment should be described>

The attribute constraint is used to describe non-functional aspects that may restrict the goal of a scenario to be achieved within the desired quality. These non-functional aspects can be related to context, resources and episodes

The existence of relationships among scenarios is an important characteristic of this representation. Scenarios can be connected to other scenarios through links, yielding a complex network of relationships. These links can be of four distinct types: constraint, precondition, *sub-scenario* and *exception*. A constraint as well as a precondition can be described by another scenario. *Sub-scenario* or *exception* relationships are defined when an ***episode*** (sentence) or ***exception*** (solution) of a scenario is detailed in another scenario.

2.1.4. Natural Language-based Scenario Representations Compared

There is no clear and correct answer when it comes to selecting the right representation or language for writing scenarios. The selection of appropriate technique depends on different factors and can be different for every project. Most of the existing languages or templates for writing use cases are extensions based on textual use case template proposed by Cockburn (2001). Therefore, most of the existing use case templates only represent specific situations between the user and the system through user interface. Other drawbacks of use cases based on Cockburn (2001) template are the following:

- Lack of precise definition, which originated that several companies have reinvented their own versions (Lee et al., 1998; Sinnig et al., 2009).

- Consider only the user interactions with the system.
- The relationships among use cases are rarely explicit.

On the other hand, the scenario language proposed by Leite et al. (2000) represents situations in the domain application. A situation describes the interactions among actors in the Universe of Discourse, including interactions with a software system (existing or a future one) or the internal behavior of the application. Besides in Leite et al. (2000), are presented powerful characteristics to make explicit the relationships among different scenarios.

Table 3 - Scenario and Use Case Comparison

Scenario (Leite et al., 2000)	**Use Case** (Cockburn, 2001)
Title	Use Case #
Goal	Goal in Context
Context	Scope & Level
	Preconditions
	Success End Condition
	Failed End Condition
Resources	*Not applicable*
Actors	Primary, Secondary Actors
	Trigger
Episodes	Description
Exception	Extensions
	Sub-Variations

Table 3 compares the elements that compose the scenario language (Leite et al., 2000) and use case representation (Cockburn, 2001). Scenario title and goal are equivalent to Use Case name and goal, respectively. Scenario context can be equivalent to Use Case scope & level, pre-conditions and conditions. Scenario episodes are equivalent to use case description element. Scenario exception can be represented by use case extensions or sub-variations because they are triggered by situations that prevent the main course of actions. Scenario resources element is *not applicable* to use cases because use case does not consider this element.

2.2. Quality in Software Requirements Specification

Software quality is defined by the IEEE as "the degree to which a system, component or process Answer: (1) the specified requirements, and (2) the expectations or needs of customers or users". On the other hand, the ISO defines quality as" the totality of characteristics of a product or service that demonstrate their ability to meet needs specified or implied". Therefore, these two definitions

show that the quality of a software product is closely linked to meeting their requirements.

In requirements engineering, requirements quality is not just about whether the functionality has been correctly documented (Quality Assurance), but also depends on non-functional requirements.

2.2.1. Non-functional Requirements (NFR)

Non-functional requirements (NFRs) are often called quality attributes of a system. Other terms for non-functional requirements are constraints, quality goals, quality of service requirements and non-behavioral requirements. In contrast to functional requirements, non-functional requirements define how a system should behave. They significantly influence the product quality of the final software system (Sommerville, 2010).

Quality attributes are hard to specify and are usually stated informally. In Goal-Oriented Requirements Engineering approaches (GORE), non-functional requirements are represented as soft-goals, whose satisfaction cannot be established in a clear-cut sense. The main objective of GORE is to iteratively refine higher-level requirements until concrete system requirements are obtained. The NFR framework developed by Chung et al. (2000) is a goal-oriented approach to represent non-functional requirements.

2.2.1.1. NFR Framework

The NFR Framework (Chung et al., 2000) is a Goal-Oriented RE approach for capturing NFRs in the domain of interest, and defining their interdependencies and operationalizations. The NFR Framework allows to: (1) model the NFRs and their decomposition, (2) design alternatives for different NFRs, (3) deal with conflicts, tradeoffs, and priorities, and (4) evaluate the decisions impact centered on NFRs. These NFRs are modeled using a Soft-goals Interdependency Graph (SIG). The SIG graphically represent NFRs as soft-goal nodes (clouds); their refinements using AND/OR decompositions links; their positive/negative inter-dependencies as some+ (help), some- (hurt), some++ (make), some-- (break) contribution links; their operationalizations as leaf nodes; and claims as annotations in natural language. Generally, soft-goals are named using the

convention Type [Topic1, Topic2...] where Type is the soft-goal and Topic is the field of application of Type; Topic is optional.

Figure 3 illustrates a very simple SIG that models the Software Requirements Specification Correctness, by considering its contributions – HELP links – in Specification Consistency, Completeness and Unambiguity. An inter-dependency between *Consistency* and *Completeness* negatively impacts (HURT) on both, because increasing Completeness might negatively impact Consistency. According to Glinz (2000), and Zowghi and Gervasi (2003), there is an important causal relationship between Consistency, Completeness and Correctness.

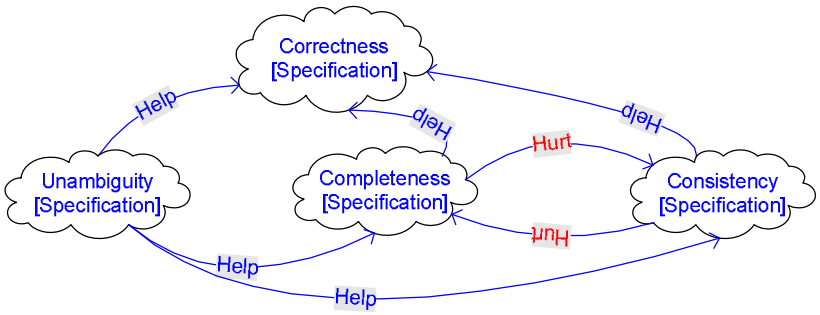


Figure 3 - SIG of Correctness.

2.2.2. Quality Assurance for Software Requirements

Evaluating the quality of a SRS involves aspects related to quality assurance of software artifacts produced in the requirements engineering process. The quality of a requirements artifact can be addressed by the use of metrics, standards, prototyping, indicators or tests. A critical review of these artifacts may be accomplished through inspections.

Various authors have worked on quality assurance of software requirements. Some focus on the classification of quality into characteristics (Gnesi et al., 2005; Wilson et al., 1997; Lucassen et al., 2015; Arora et al., 2015), and the definition of quality models and metrics to quantify the presence of evidences (or not) of these characteristics; others developed comprehensive checklists or constructive approaches (Glinz, 2000; Leite et al. 2000; Denger et al., 2005; Phalp et al., 2007; Sinha et al., 2010) for detecting defects classified into taxonomy of defects.

Some researches focused in the automatic detection of defects in requirements (Gnesi et al., 2005; Arora et al., 2015; Sinha et al., 2010), taking advantage of NLP techniques.

According to Gnesi et al. (2005), a *Quality Model* is the formalization of the definition of the term "quality" to be associated to a type of work product. The typical objectives of a quality model are to define, analyze, and document a product's:

- *Quality Characteristics*: define and document the relevant quality factors (also known as quality attributes or "ilities") that are important attributes of work products (e.g. applications, components, or documents) or processes that characterizes part of their overall quality (e.g. extensibility, operational availability, performance, re-usability, …). *Quality sub-characteristics* are important properties of quality characteristics.
- *Quality Indicators*: are specific descriptions of something that provides evidence either for or against the existence of a specific quality characteristic or sub- characteristic.
- *Quality Metrics*: provide numerical values estimating the quality of a work product or process by measuring the degree to which it possesses a specific quality characteristic.

2.2.2.1. Software Requirements Quality Characteristics

Most of the quality models or checklist for software requirements specifications (detailed in Section 2.5) are based on the quality characteristics defined by the IEEE std 830-1998 standard (IEEE, 1998). Among others, the desirable characteristics of a "good" SRS are:

- **Complete**: An SRS is complete if, most of 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).
- **Consistent**: An SRS is consistent when, two or more requirements are not in conflict with one another (Boehm, 1979).
- **Correct**: An SRS is correct if and only if, every requirement stated therein is one that the software shall meet (IEEE, 1998).
- **Unambiguity**: An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation (IEEE, 1998). *Unambiguity*

> requires the specification to be as formal as possible; however, in the vast majority of requirements specifications, requirements are stated informally with natural language or at best semi-formally. Thus, unambiguity is very difficult to achieve. (Glinz, 2000).

The ISO 29148 (IEEE, 2011) standard was created to harmonize a set of existing standards or quality models, including the IEEE 830-1998 (IEEE, 1998) standard. It differentiates between quality characteristics for a *set of requirements*, such as completeness or consistency, and quality characteristics for *individual requirements*, such as ambiguity or singularity. Apart from the "dos", it also provides some "don'ts" regarding requirements language (e.g. avoid ambiguous adverbs, vague pronouns, subjective language, and so on). Some researches classify these "don'ts" as indicators or smells of ambiguity (Wilson et al., 1997; Gnesi et al., 2005; Tjong, 2008; Femmer et al., 2014).

In practice, most software requirements specifications do not meet these quality characteristics. Thus, it is impossible to be complete as well as to assure correctness due to the completeness fallacy (Leite, 2007).

2.2.2.2. Verification & Validation

The terms Verification and Validation are commonly used in software engineering to mean two different types of analysis. According to Boehm (1979), the usual definitions are:

- **Verification**: to establish the truth of the correspondence between a software product and its specification, i.e. are we building the product right?
- **Validation**: to establish the fitness or worth of a soft-ware product for its operational mission, i.e. are we building the right product?

In other words, *validation* is concerned with checking that the software meets user's actual needs, while *verification* is concerned with whether the software is well-engineered, error-free, and so on. Verification helps to determine whether the software is of high quality, but it does not ensure that it is useful (Easterbrook, 2010).

Verification is a relatively objective process. It includes the activities associated with producing high quality software: inspection, analysis, simulation

or checklist heuristics for evaluating that specifications are expressed precisely enough.

In contrast, validation is a subjective process. It should confirm that the Universe of Discourse situations, occurrences, have been reported in accordance with the real world needs of the users. Requirements validation includes techniques such as mock-up, storyboards and high level prototyping, and must be performed with clients and users.

A SRS can be verified by the following approaches: (1) *inspection*, to examine carefully and critically, especially for flaws; (2) *analysis*, a series of logical deductions based on logic or math oriented representations; (3) *simulation*, execution of a model, usually with a computer program; (4) *checklist*, an examination of the SRS by pre-defined rules, patterns or taxonomies.

Each of the above mentioned approaches for SRS verification has its own advantages and drawbacks. Formal verification through model checking techniques can be used for analysis of structural and behavioral properties; however, this approach is not well suitable for models with large number of states (the complexity of the generated reachability graph is exponential).

2.2.2.3. Quality of Scenarios

Many authors have suggested using guidelines for writing use case descriptions (e.g. Cockburn, 2001; Denger et al., 2005; Phalp et al., 2007) and such guidance is often entirely plausible. For example, Cockburn's (2001) recommendation of "subject...verb... direct object... prepositional phrase", appears to be particularly straightforward and intuitive. Although these structure guidelines are meant to aid composition, the ultimate goal is to improve the resulting description.

Similarly, other authors suggest to measure compliance with the guideline suggestions (e.g.; Anda and Sjøberg, 2002; Denger et al., 2005; Leite et al., 2005; Phalp et al., 2007; Sinha et al., 2010). Indeed, these proposals use inspection techniques by examination of scenario (or use case) descriptions, and find defects using checklists.

Leite et al. (2005) developed a scenario-based reading (inspection) technique to improve the qualities of scenarios. The technique allows identifying

missing, incorrect, ambiguous, contradicting and overlapping information in scenarios. The output is a list of discrepancies, errors, and omissions (DEOs). The inspection process is divided into four steps: plan, prepare, meet and rework. Their method was applied in 9 different case studies and returned positive results.

It was difficult to find formal definitions of quality characteristics related to scenarios; most of the authors suggested the use of taxonomy of defects and checklists to find potential problems. Anda et al. (2009) improved a previous use case inspection technique (Anda and Sjøberg, 2002), by defining an initial model of quality attributes for UML use cases.

2.3. Concurrency

Concurrent systems (i.e., ones where there is more than one process existing at a time) present characteristics such as ***non-deterministic*** and ***synchronization*** between processes. Web systems provide the most obvious examples of concurrent systems, which can be characterized as a system where there are a number of different processes being carried out at the same time.

According to Roscoe (1998), what all concurrent systems have in common is a number of separate processes which need to interact with each other. Therefore, the crucial thing which makes concurrent systems different from sequential ones is the fact that their processes interact with each other *at the level of* ***communication*** (Roscoe, 1998). To understand this point, one process communicates with another by a named ***communication channel*** or a ***shared resource*** (or variable).

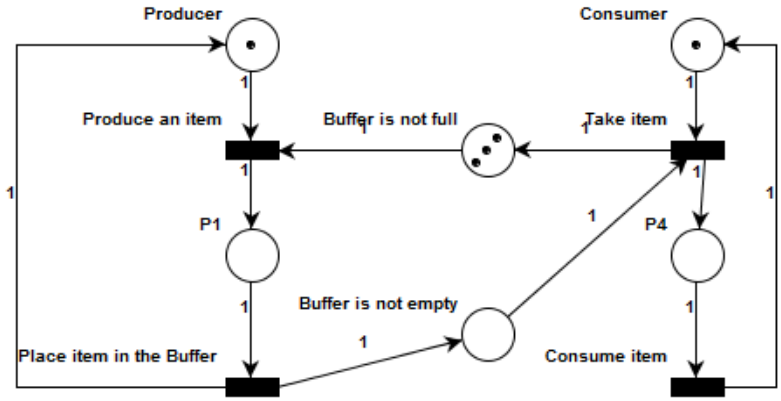


Figure 4 – Producer and Consumer Problem Using Petri-Nets

The *Producer-Consumer* problem is an example of communicating processes using a named channel called "***the buffer***"; the producer sends a

message to the consumer putting an element in the buffer. Figure 4 shows this communication using a Petri-Net model.

In the *Readers-Writers* problem, processes communicate with each other by a ***shared resource***; while the *shared resource* is being written or modified by the writer process, it is often necessary to bar other writer or reader processes. Figure 5 shows this communication using a Petri-Net model.

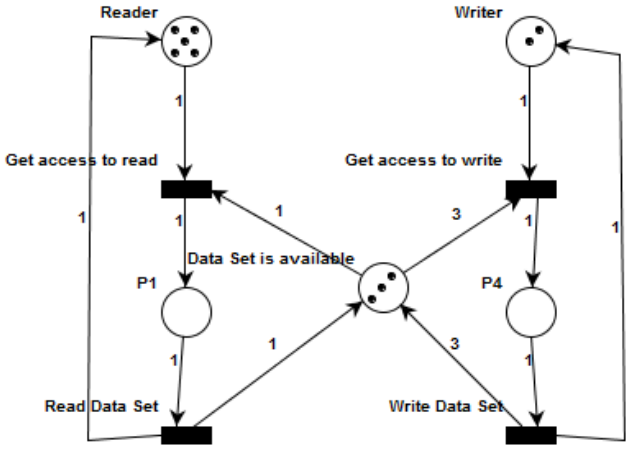


Figure 5 – Reader and Writer Problem Using Petri-Nets

2.3.1. Synchronization

According to Downey (2005), ***Synchronization*** refers to relationships among several processes and any kind of relationships (before, during, after), i.e. it is related to the execution order of processes. *Synchronization* between two processes means that one process necessarily waits the other process. In sequential systems, computers execute one process after another, and it is possible to know the order of execution; but in concurrent systems it is impossible to tell. Examples of synchronization include: (1) *Serialization* - event A must happen before Event B; and (2) *Mutual exclusion* - events A and B must not happen at the same time.

2.3.2. Non-determinism

A system exhibits ***Non-determinism*** if two different copies of it may behave differently when given exactly the same inputs (Roscoe, 1998). Concurrent systems are often non-deterministic because of interleaving accesses on shared resources, i.e. concurrent systems suffers from limited ***controllability*** and ***observability*** problem.

2.3.3. Synchronization Constraints

Due to *non-deterministic* behavior, it is difficult to ensure that the system is free of *synchronization* bugs. The most common alternative to reduce these bugs is the use of synchronization constraints to control concurrent access to *shared resources*. *Serialization* and *Mutual exclusion* are examples of synchronization constraints. Serialization may seem trivial, but the underlying idea, ***message passing***, is a real solution for many synchronization problems (Downey, 2005).

The most common constraint is *mutual exclusion*, or *mutex*; *mutex* guarantees that only one process accesses a shared resource at a time, eliminating the kinds of synchronization bugs. Like serialization, mutual exclusion can be implemented using ***message passing*** (Downey, 2005).

2.3.4. Desired Properties of Concurrent Systems

A system should employ the principles of modularity (top-down design) and make explicit the interconnectivity among modules. Modularity is considered as a mechanism to deal with the complexity of concurrent systems (Lee et al., 1998).

The design of concurrent systems is often modular and proceeds by first developing local processes, services, modules or components, and then composing these modules to one component. While the composing is performed using synchronization constraints, it is possible to introduce some concurrency bugs that impair the following desired properties of concurrent systems.

2.3.4.1. Deadlock-free

A system should not have any deadlock situation. A concurrent system is *deadlocked* if no process can make any progress, generally because each is waiting for communication with others (Roscoe, 1998).

2.3.4.2. Boundedness

This property refers to the limited capacity of a communication channel or a shared resource (Murata, 1989). A concurrent system is overflowed when the number of elements in some channel or resource exceeds a finite capacity.

2.3.5.
Petri-Net

This section presents the fundamentals of Petri-Nets, especially of place-transition Petri-Nets (Reisig, 1985; Murata, 1989).

Petri-Net is a graphical and mathematical language for modeling and analysis of systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. Due to these features, Petri-Nets can be used for modeling and analysis of: performance, communication protocols, distributed-software systems, distributed-database systems, concurrent and parallel programs, industrial control systems, discrete-events systems, multiprocessor memory systems, dataflow-computing systems, fault-tolerant systems.

They were introduced by Carl Adam Petri in 1962 at the Technische Universität Darmstadt, Germany.

A Petri-Net (Figure 6) is a directed, weighted, bipartite graph; and it is composed of nodes that denote places (Place) or transitions (Transition). Nodes are linked together by arcs (Arc).

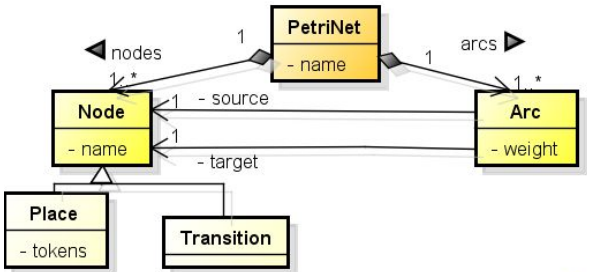


Figure 6 - Petri-Net metamodel (Sarmiento et al., 2015)

2.3.5.1.
Petri-Net Definitions

Transitions are active components. They model the activities 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) for the activity have been fulfilled.

Places are passive components and placeholders for tokens. They model communication medium, buffer, geographical location or a possible state (condition). The current state of the system being modeled is called *marking*, which is given by the number of tokens in each place.

Tokens model physical or information object, collection of objects, resource availability, jobs to perform, flow of control, synchronization conditions, indicator of state or indicator of condition.

In addition, *tokens* are used in Petri-Nets to simulate the dynamic and concurrent activities of systems.

Arcs are of two types. Input arcs start from places and ends at transitions, while output arcs start at a transition and end at a place.

Definition 2.1. A ***place-transition*** Petri-Net (Reisig, 1985) is a five-tuple $PN = (P, T, F, W, M_0)$ where $P = \{p_1, p_2, ..., p_n\}$ is a finite set of places, $T = \{t_1, t_2, ..., t_m\}$ is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $W : F \rightarrow \{1, 2, ...\}$ is a weight function, $M_0 : P \rightarrow \{0, 1, 2, ...\}$ is the initial marking and $P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

In addition to have a static structure defined above, systems change over time and it is of great interest to study its dynamic behavior. In Petri-Nets, *markings* represent the states of the system over time. Figure 7 shows an example of a marked Petri-Net.

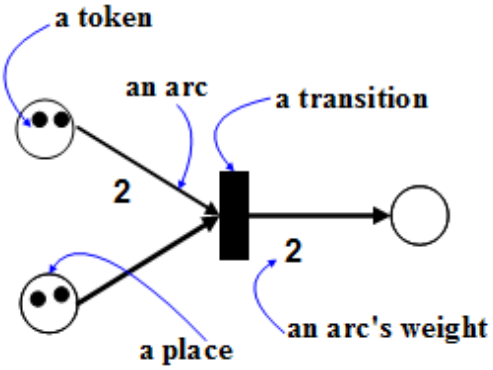


Figure 7 - Marked Petri-Net

Definition 2.2. For a $PN = (P, T, F, W, M_0)$, a ***marking*** is a function $M: P \rightarrow \{0, 1, 2, ...\}$, 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 some at all of its output places. The number of tokens removed/added depends on the *cardinality* (*weight*) of each arc. Figure 8 shows an example of a marked Petri-Net with a transition enabled for firing.

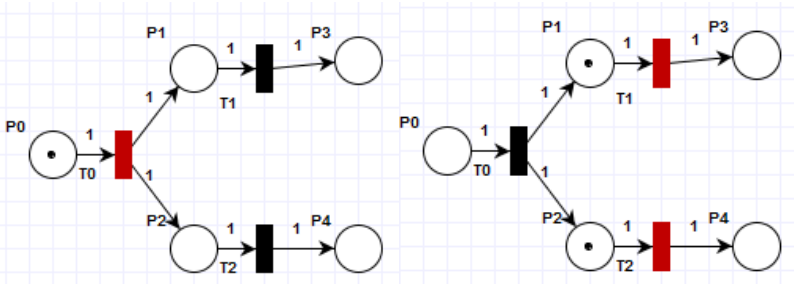


Figure 8 - (a) Transitions before Firing, (b) Transitions after firing

Definition 2.3. A transition *t* is ***enabled*** for firing at a marking *M* if $M(p) \geq W(p, t)$ for any $p \in {}^{o}t$ where ${}^{o}t$ 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*. t^{o} is the set of output places of *t*. The notations ${}^{o}p$ or p^{o} have the same meaning for places.

Definition 2.4. For a *PN*, a sequence of transitions $\sigma = \langle t_1, t_2, ..., t_n \rangle$ is called a ***firing sequence*** if and only if $M_0\ [t_1\rangle, [t_2\rangle, ..., [t_n\rangle\ M_n$. In notation, $M_0\ [PN, \sigma\rangle\ \underline{M_n}$ or $M_0\ [\sigma\rangle\ M_n$.

2.3.5.2. Modeling with Petri-Nets

In the real world, events may happen at the same time changing the state of a system over time. In systems modeled as Petri-Net models, the *states* of the system changes via *enabling* and *firing transitions*. A system may have many local states to form a global state.

Due to complex behavior of the systems, there is a need to model the relationships among several events of the system. In Petri-Net models, we may describe these relationships using sequential, non-deterministic, concurrency and synchronization structures.

In a sequential structure, events happen in a sequential order. In Figure 9, event *t0* fires before *t1*, *t1* fires before *t3*.

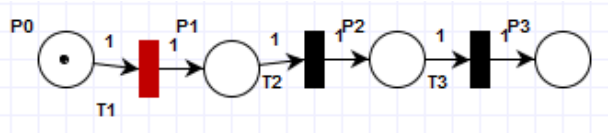


Figure 9 - Sequential structure

In a non-deterministic (conflict, choice, decision) structure, only one of the simultaneously enabled events may happen. In Figure 10, only one of event *t1* or *t2* may fire.

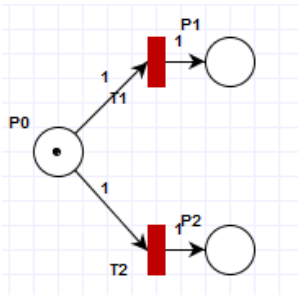


Figure 10 - Non-deterministic structure

In a concurrency structure, all simultaneously enabled events may happen. In Figure 11, events *t1* and *t2* may fire simultaneously.

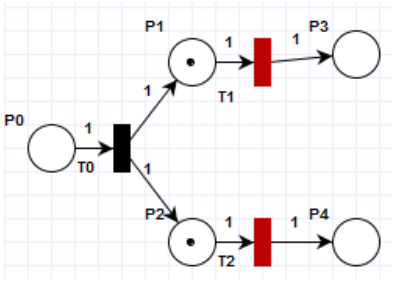


Figure 11 - Concurrency structure

In a synchronization structure, an event happens only if all events defined as pre-conditions or inputs happen. In Figure 12, event *t3* fires after events *t1* and *t2* fire.

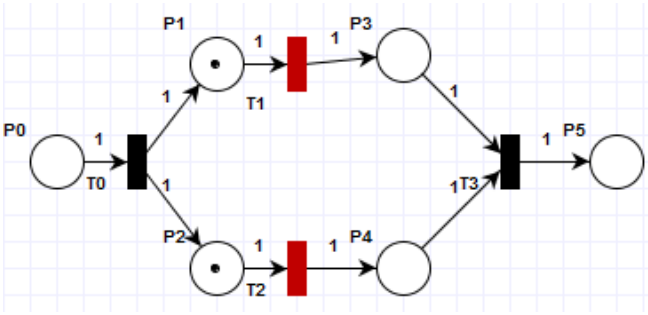


Figure 12 - Synchronization structure

In complex systems, several events happen at the same time and interact with each other. These relationships among events can lead to problems or erroneous situations such as deadlocks.

2.3.5.3. Analysis of Petri-Nets

One feature that makes Petri-Nets interesting is that they also provide the capability to analyze model properties. The analysis of Petri-Net models evaluates defects related to structural and dynamic properties. The structural properties can

be detected traversing the flow relation between places and transitions, while dynamic properties can be detected using the *initial marking* and *markings* which can be reached by firing transitions. S*imulation, reachability/coverability* or *invariant* analysis are methods for detecting defects due to dynamic properties like reachability, boundedness, liveness, and deadlock free (Reisig, 1985).

The most important analysis strategy is the *reachability* (Murata, 1989). Figure 13 shows that the marking of the Petri-net in Figure 13 (b) is reachable from the marking in Figure 13 (a).

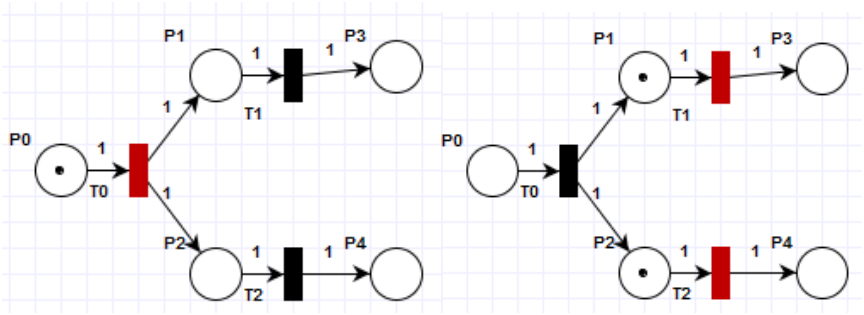


Figure 13 - (a) Transitions before Firing, (b) Transitions after firing

Definition 2.5. For a $PN = (P, T, F, W, M_0)$, a marking M is said to be ***reachable*** if and only if there exists a firing sequence σ such that $M_0 [\sigma\rangle M$. In notation, M_0 [PN,*⟩ M or M_0 [*⟩M. represents the set of all reachable markings of *PN*.

If a Petri-Net is to be a model of a real hardware device, one of the important properties it should have is *safeness***.** *Safeness* is a special case of the property called *Boundedness***.**

Definition 2.6. 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. A *PN* is ***safe*** if it is *1-bounded*.

Other important property is the concept of liveness (when modeling operating systems); and it is closely related to the complete absence of deadlocks (Murata, 1989).

Definition 2.7. 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 $\forall M \in [M_0\rangle, \exists M' : M[*\rangle M'[t\rangle$. *PN* is said to be ***live*** if and only if every transition is ***live***.

Definition 2.8. 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 dead marking of *PN*, i.e., a deadlock.

A Petri-Net *PN* is called ***deadlock-free*** if deadlock does not exist in PN. $\neg M\ [t\rangle$ denotes that *t* is disenabled under *M*.

Another desirable property is that the system (when modeling manufacturing systems) can be re-initialized from any reachable state.

Definition 2.9. A *PN* is ***reversible*** if M_0 is reachable from each other reachable marking *M*.

The reachability analysis method generates a reachability graph which contains reachable *markings* as nodes and *transitions* as arcs (which effect the change from one marking to another by firing). We can get an overview about possible states. Figure 14 shows the reachability graph for Petri-Net depicted in Figure 12 using the PIPE2 (2015) tool.

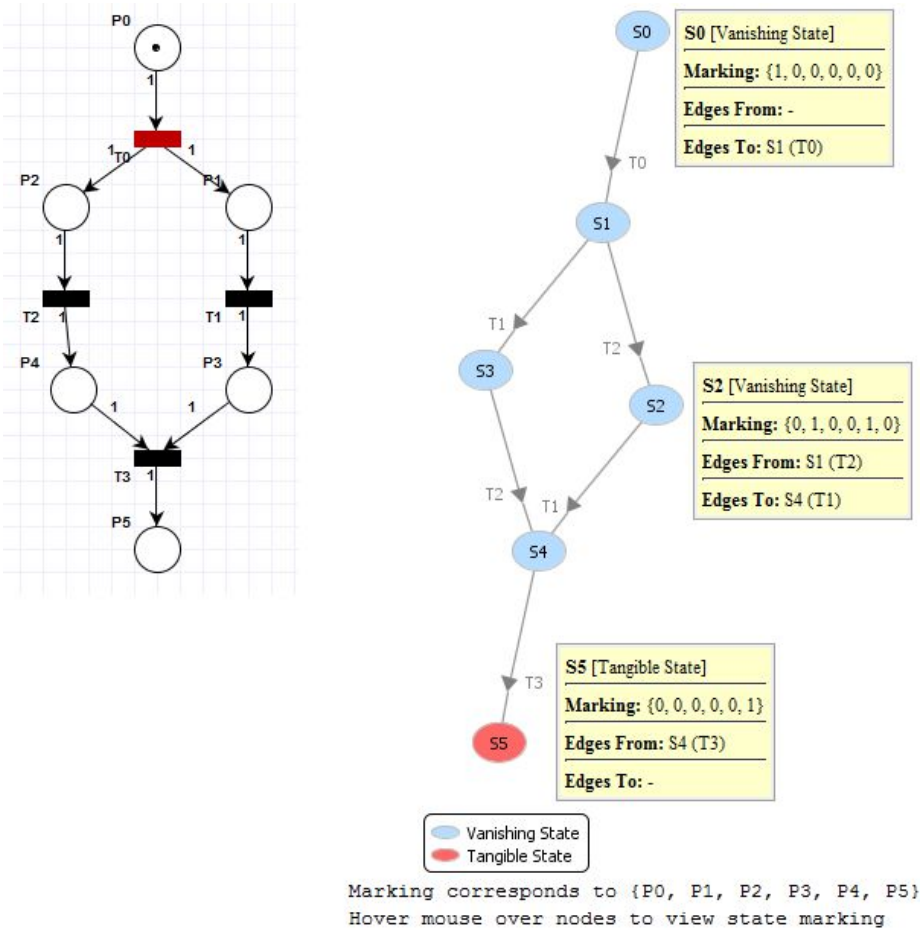


Figure 14 - A Reachable Petri-Net (generated using PIPE2, 2015)

In case of Petri-Nets with infinite many markings the computation of the reachability graph with this method fails. The state explosion issue is a serious problem when applying Petri-Net analysis to large systems, In fact, the generation of the reachability graph of a Petri-Net model requires an exponential space. Research continues to work on how to do it efficiently.

2.4. Considerations about Scenarios and Concurrency

Such as introduced by Lee et al (1998) and Leite et al (2000), scenarios are rarely truly independent in practice; they may interact by complex relationships, what can lead to erroneous situations such as conflicts. Some of these relationships include potentially concurrent scenarios.

Considering that some languages for writing scenarios present characteristics for representing or identifying the relationships among different scenarios; they can be used to perform early concurrent system reliability assessment, i.e. scenarios can be used to detect potential concurrency defects at early software development activities.

Thus, scenarios are a key concept for writing software requirements, because: (1) they describe requirements such that users and developers can easily understand; (2) they may make explicit the relationships between scenarios, such as introduced by Leite et al (2000); and (3) they may support the early detection and resolution of quality defects.

Due to these characteristics, a scenario-driven requirements engineering approach has the potential to influence positively the Software Development process. Requirements engineers need to pay special attention to its quality, in special with respect to ***unambiguity***, ***completeness***, ***consistency*** and ***correctness*** since they will anchor further development.

2.5. Related Work

This sub section: (1) discusses research (or tools) related to the analysis of Natural Language based Software Requirements Specifications; (2) compares techniques to analyze static and dynamic aspects related to quality attributes of requirements; and finally (3) research gaps are appointed.

2.5.1. Analysis of Software Requirements Specification

Not all the aspects related to the analysis of *SRS* quality can be addressed in the same way and with the same depth and with the same ease, because it depends not only on internal requirements specifications, but also on external

specifications (domain information or other software artifacts) and actual user's satisfaction.

In fact, **Correctness** evaluation of SRS, i.e. the verification that the system to be constructed is correctly described by them, needs to be supported by more rigorous methods (Glinz, 2000), and it depends on actual user's needs satisfaction.

Fortunately, some properties related to static (structural) and dynamic (behavioral) aspects of SRS can be addressed without increasing the formalism level. These properties may be grouped into three principal groups: **Unambiguity, Completeness** and **Consistency**.

Several studies dealing with the evaluation and the achievement of quality in NL based SRS can be found in the literature. Most of them are based on the definition of taxonomy of common defects or checklists necessary to improve the main quality characteristics of software requirements (unambiguity, completeness, consistency and correctness). Effectively, checklists or taxonomy of defects are used to perform the analysis of NL requirements aiming to detect defects and collect metrics. We will discuss some literature that we consider to be of particular interest to our research.

2.5.2. Overview of the State of the Art

Quality in NL requirements can be addressed through the use of two types of analysis techniques, or the combination of them:

- **Static Analysis**: Analysis of the style and content of individual requirements or a set of requirements. This is accomplished by verification of conformance to the representation language, and in some cases by NLP techniques that can make explicit ambiguous terms and styles.
- **Dynamic Analysis**: Analysis of the behavior of a set of requirements to identify inconsistency or incorrectness defects, such as conflicting or overlapping requirements.

2.5.2.1. Static Analysis of Software Requirements Specification

Requirements written using natural language are usually hard to analyze, because natural language is by definition ambiguous. Consequently, the inherent

ambiguity of natural language makes difficult the evaluation of properties related to completeness and consistency quality attributes.

Ambiguity analysis and checking for properties related to **completeness** and **consistency** in requirements are usually performed by different stakeholders (e.g., developers, testers, customers, project managers) through a tedious procedure of reading requirements documents and looking for defects. According to Gnesi et al. (2005), most of the defects found by inspection techniques are in fact simple linguistic defects.

In static analysis, it is not necessary to execute the requirements for ***searching defect indicators that hurt unambiguity and completeness***. Static analysis techniques may provide support that can facilitate the work of requirements engineers for checking consistency.

Software requirements (user or system requirements) may be a set of statements or scenarios; thus, static analysis techniques may be applied on ***requirements*** described as ***statements*** or represented as ***scenarios***, depending on the level of abstraction.

2.5.2.1.1. Static Analysis of Requirement Statements

Requirement statements are high-level descriptions of system functionalities. Most of the existing studies about analysis of **requirement statements** are addressed for detecting ambiguity indicators within individual statements. Only a few works (Arora et al., 2015; Lucassen et al., 2015) were proposed for searching defect indicators that contributes to completeness (mainly, conformance to requirements **templates**).

Wilson et al. (1997) examine the quality evaluation of NL software requirements. This approach defined a quality model composed of quality attributes and quality indicators, and develops an automatic tool called ***Automated Requirement Measurement*** (ARM) to perform the analysis against the quality model aiming to detect defects and collect metrics by searching the SRS document for words or phrases that have been identified as quality indicators, e.g. weak phrases. The quality model is composed of desirable characteristics for requirements specifications (*Complete, Consistent, Correct, Modifiable, Ranked, Traceable, Unambiguous* and *Verifiable*); although most of these quality attributes

are subjective, there are aspects that can be measured and are indicators of quality attributes.

Based on the analysis of a set of requirements specification documents of NASA projects, nine categories of quality indicators were established. These categories are related to the evaluation of individual specification statements and entire requirements documents. Individual indicators were identified by finding *frequently used words*, *phrases*, and *structures* of the selected documents that were related to quality attributes and could be easily identified and counted by computer programs. These individual indicators were grouped according to their *indicative characteristics*.

Table 52 (Appendix 2) shows the main categories of indicators (or properties) that hurt *Unambiguity* and, for each of them, a sub-set of indicators detected by the ARM tool.

Gnesi et al. (2005) define a quality model for Natural Language requirements specifications, composed of the three target qualities to be achieved (*Expressiveness*, *Consistency* and *Completeness*). They concentrate on *Expressiveness-related* issues, leaving *consistency* and *completeness* problems for further studies. It includes those quality characteristics dealing with the understanding of the meaning of the requirements by humans. Linguistic techniques were used to address the issues related to the *Expressiveness* because the lexical and syntactical levels provide means enough to obtain effective results.

Expressiveness quality model is composed of three quality characteristics (Unambiguity, Specification Completion and Understandability) to be evaluated by means of indicators. We understand that these three quality characteristics are related to *Unambiguity* main quality. Indicators are linguistic components of the requirements directly detectable and measurable on the requirements document (See Table 53 in Appendix 2) that hurt *Unambiguity*.

In this approach the ***Quality Analyzer for Requirements Specifications*** (QuARS) tool performs a lexical analysis (morphological analysis) of the requirements document to evaluate properties that hurt Unambiguity. *Vagueness*, *Optionality, Subjectivity* and *Readability* defects are detected based on the occurrence of special *terms* in the requirements or the number of elements in the sentences. To point out the other indicators, it performs a syntactical analysis

because the knowledge of the syntactical structure of the sentences is required to detect an *Implicitly*, *Weakness*, and *Multiplicity* indicators.

Tjong (2008) proposes an approach for **Avoiding Ambiguity in Requirements Specifications**, which provides a set of guideline rules and an inspection checklist for writing less ambiguous requirements. The guideline rules and checklist were implemented in an experimental lexical analyzer tool called **SREE**. During the inspection of the NL Requirements, it notifies the user about the potential ambiguity in the document, leaving space for the user to act upon and disambiguate a truly ambiguous statement.

The SREE is based on an Ambiguity Indicator Corpus (AIC), which contains the corpus of indicators of potentially ambiguous keywords, key phrases, and symbols. Although it may not be possible to have an AIC that contains an indicator of every possible potential ambiguity due to the richness of NL, SREE allows its user to add new indicators to its AIC. There are two categories of AICs in SREE, the Original Indicator Corpus (OIC) and the Customized Indicator Corpus (CIC). The OIC contains ten categories of ambiguity indicators (*corpi*), each in a separate file and each named appropriately for the nature of the potential ambiguities indicated by its contents: *Continuance, Coordinator, Directive, Incomplete, Optional, Pronoun, Plural, Quantifier, Vague,* and *Weak*. Each of these *corpi* has its own list of indicators. SREE automatically loads these *corpi* into the AIC each time a user starts up SREE.

Table 54 (Appendix 2) shows some of these indicators of these *corpi*, organized as properties that hurt Unambiguity.

In essence, SREE is a lexical analyzer. When SREE finds a word in its input matching one of the indicators in its AIC, then SREE notifies its user by printing out a message describing the kind of potential ambiguity suffered by the word.

Femmer et al. (2014) study the application of light-weight static analyses to make instant **checks on natural language requirements and detect bad smells**. Based on the ISO/IEC/IEEE 29148 standard (ISO, 2011), they derive a set of 8 smells that indicate potential issues in requirements specifications: *ambiguous adverbs and adjectives, vague pronouns, subjective language, comparative phrases*, and so on (See Table 55 in Appendix 2).

The authors implemented a prototype tool for detecting patterns that are signs of potential quality defects in requirements by using NLP techniques, such as, part-of-speech tagging, morphological analysis, and customized dictionaries.

However, the granularity of the smells identified by the tool is limited to individual sentences, disregarding duplicate functionality among documents.

Arora et al. (2015) present an automated and tool-supported approach for **checking conformance to requirements templates** (conform to the templates of Rupp's (Pohl and Rupp, 2011) and EARS (Mavin et al., 2009). The approach builds on a mature Natural Language Processing technique, known as *text chunking*. In this context, the approach builds an NLP pipeline for text chunking. To instantiate the pipeline, one needs to choose, for each step in the pipeline, a specific implementation from the set of existing alternative implementations.

In addition to checking template conformance, they use NLP for detecting and warning about several potentially problematic constructs, also called requirements smells (Femmer et al., 2014), that may be signs of vagueness or ambiguity in requirements statements. Table 56 (Appendix 2) lists and exemplifies several constructs that can be detected automatically.

Lucassen et al. (2015) propose a Quality User Story Framework with 14 different criteria (See Table 57 in Appendix 2) to assess the quality of user stories (conform to the format of Cohn, 2004). Furthermore, the researchers developed a conceptual model to improve the quality of raw user stories by exploring defects and deviations in user stories. They developed the ***Automatic Quality User Story Artisan*** (AQUSA) tool on a NLP technique, which chunks the text and exposes defects and deviations from good practice in user stories. Unlike most NLP tools for RE, and in line with Berry's notion of a dumb tool (Berry et al., 2012); the tool detects some defects with close to 100% recall and high precision; this is necessary to avoid that a human requirements engineer has to double check the entire requirements document for missed defects. Consequently, AQUSA can support only *certain syntactical* and *pragmatic criteria* in an effective manner.

2.5.2.1.2. Static Analysis of Scenarios

Scenarios are described by a set of more detailed statements of the functionality to be performed. Usually, **scenarios** are written using natural

language. In our work, **scenario** and **use case descriptions** are considered **synonymous** because they are described by similar components.

Most of the existing approaches for scenario-based representations analysis are focused on checking of properties related to completeness, i.e., checking the conformance to scenario templates or guidelines and the coherence among scenario components (e.g. actors, pre-conditions, main flow descriptions, alternatives or extensions).

Only a few works (Leite et al., 2000; Anda and Sjoberg, 2002; Sinha et al., 2010, Liu, 2015) focused on checking some properties from the relationships among different scenarios. In these approaches, related scenarios are explicitly referenced within statements described in scenario descriptions.

Anda and Sjoberg (2002) develop a taxonomy of defects in use case diagrams and use case descriptions (conform to the template of Cockburn, 2001) considering different stakeholders. The defects are divided into omissions, incorrect facts, inconsistencies, ambiguities and extraneous information. They proposed a checklist-based inspection technique for detecting such defects in: (1) actors and use cases of use case diagrams; and (2) flow of events, variations, relationships between use cases, triggers, pre-conditions and post-conditions of use case descriptions. Table 58 (Appendix 2) shows the defects detected by the inspection technique.

Leite et al. (2005) present a strategy for **scenario inspections** to be performed by requirements engineers as a verification process before validation with clients, and they show how inspections help software developers to better manage the production of scenarios. This strategy was designed to be integrated with a specific **scenario construction process** (Leite et al., 2000) and was applied to several case studies. The data collected regarding the types of problems support their claim that scenario inspections do improve scenario quality.

Table 59 (Appendix 2) shows a checklist with verification heuristics for looking for discrepancies, errors and omissions (DEOs) in scenarios and their relationships with other scenarios and the symbols of the Language Extended Lexicon (LEL). The LEL (Leite et al., 2000) registers symbols (words or phrases) that are peculiar to the application domain. The types of symbols are: Subject, Object, Verb and State. Most of the heuristics for checking may be automated by intelligent editors and verification agents.

Phalp et al. (2007) describe a Use Case Description Quality Checklist that acts as a check on the quality of the written description for detecting defects in use case descriptions. They proposed a set of heuristics (***7 Cs of Communicability***) that organizes the ideas, comments and suggestions of related work into categories relevant for improving the communicability of use case descriptions, and consequently producing a coherent set of desirable use case qualities. Their results of one experiment indicate that when a checklist is used in inspections it is mostly errors in syntax that will be discovered because they are easier to find than semantic ones.

The ***7 Cs*** quality model is proposed as a set of heuristics to evaluate internal elements of use case descriptions (See Table 60 in Appendix 2).

Ciemniewska and Jurkiewicz (2007) developed a method for detecting requirements defects in an automatic way using simple heuristics and NLP. This thesis focuses on requirements in a form of use cases, as they consist of simple structure sentences, which are easy to analyze with available Natural Language Processing tools. Defects are organized in three levels: at the level of specifications, use cases, and steps. The level of specifications considers the behavior duplication. The level of use cases considers use cases very short or long, or complex extensions (alternative flows), among others. The level of steps considers complex syntactic structures, or omission of actors, among others. Heuristics detect defects by counting sentences, searching keywords or terms stored in dictionaries. NLP tool is used for searching verbs, subjects and objects in sentences. Table 61 (Appendix 2) shows the defects detected by the automated method.

Sinha et al. (2010) present a tool-supported approach for inspection (during edit time) of use case models (conform to a use case description metamodel), in conjunction with models from associated use cases, and reports on problems found. **Text2Test tool detects problems related to style and content of use cases** performing a linguistic analysis. Domain specific knowledge is needed to introduce semantic information to the analysis, assigning classification confidence to the concepts and verbs (type of action) described in use case sentences.

Tex2Test runs a set of checks for looking for defects in use case models; however, not every condition of interest (check) for a use case can evaluated automatically, but the set of interesting conditions that can be evaluated

automatically is quite large (if not open-ended). Some examples of these conditions are shown in Table 62 (Appendix 2), and some of which are of particular interest for test-case generation.

Liu et al. (2014) present a tool-supported approach to achieve **automatic defect detection in use case documents** by leveraging on advanced parsing techniques. In this approach, they first parse the use case document using dependency parsing techniques; the dependency parsing provides richer syntactic details, i.e., provides the subject, object and main verb information of a sentence directly. The parsing results of each use case are further processed to form an activity diagram. Lastly, they perform defect detection on the activity diagrams.

In order to find defects in a set of use cases, Liu (2015) 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 in a whole use cases graph. Traversing this graph is possible to find potential *missing scenarios* or whether a certain *pre-condition* is satisfied by certain *post-condition*.

The heuristic for finding relationships among use cases depends on an active learning strategy; they discover missing scenarios by generating questions to users. They base their approach on common use case defects shown in Table 63 (Appendix 2).

2.5.2.2. Dynamic Analysis of Software Requirements Specification

In order to improve the correctness and consistency of systems described as informal requirements (or semi-formal), it is necessary to perform an extensive and iterative analysis, which is mostly performed manually, requiring a great effort and taking a lot of time. Inexperienced inspectors often do not detect these defects or only with much effort. The other option is write requirements using formal languages, such as Petri-Nets (Murata, 1989), LTS (Keller, 1976) or CSP (Roscoe, 1998), which will allow an automatic and rigorous analysis, which usually reduces the effort and time to be done.

In order to evaluate dynamic aspects of requirements, it is necessary to execute (or simulate the execution) a set of requirements for detecting defects that can hurt properties related to **consistency** or **correctness**. The use of formal

techniques make it possible to perform a rigorous analysis and improve the consistency and correctness of a set of requirements, i.e., the set of requirements contains less erroneous situations such as conflicts or overlaps raised from the complex relationships among requirements. There is an interaction (or relationship) when two or more requirements have some effect on each other.

Many researchers have shown the importance to formalize the informal aspects of requirements in order to benefit from automated analysis of dynamic aspects. Usually, these approaches describe requirements as scenario representations. Some research focused on developing formal syntax and semantics for scenario representations, like Hsia et al. (1994) and Cheung et al. (2006); others are focusing on developing techniques to transform scenarios into executable representations. Therefore, these researches demonstrate that informal or semi-formal requirements cannot be used for further automated analysis.

Due to the focus of this work is the analysis of scenarios written using natural language, we selected related studies that focus on natural language-based scenarios. A few of them are supported by full automatic tools.

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. To facilitate the transformation, use cases are described in relation to formal definition of pre and post-conditions, and represented like Action-Condition tables. Use cases are considered as a collection of interacting and concurrently executing units of system functionalities. Petri-Net analysis techniques can be used to evaluate **completeness** and **consistency** related properties in CMPNs (See Table 64 in Appendix 2). It is the unique approach that manages the state explosion problem of Petri-Nets by dividing the CMPN into a set of slices. However, intermediate models are created and it uses a non-standard use case model without alternative/exception flows.

Lee et al. (2001) present an approach to analyze use cases using formal semantics of *Time Petri-Nets*. Use case are used to elicit system requirements; in order to represent the interaction between the actors and the system, scenarios are derived from these use cases and represented as sequence diagrams. From these sequence diagrams, Time Petri-Nets are derived to check the acquired scenarios by indicating *missing information* (incompleteness) or *wrong information* (inconsistency) hidden in these scenarios. The approach avoids deadlock

situations in the mapping process because sequence diagrams are constrained by the time line. However, relationships among use cases are no considered. **Table 65** (Appendix 2) shows the faults detected by a CASE support tool.

Denger et al. (2005) present an integrated approach for achieving high quality in use cases that combines Use Case creation guidelines, Use Case inspections, and simulation in a systematic way. They base their combined approach on a defect classification for use cases (Table 66 in Appendix 2). This classification 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 inspections are suitable for detecting subtle logical defects. *Simulation* is integrated so that serious *consistency* and *correctness* defects resulting from the interference between Use Cases can be efficiently detected; for it Use Case are mapped into *Statecharts* (Harel, 1987), and several statecharts can be simulated simultaneously.

Ad-hoc recommendations, guidelines and checklists are used for avoiding defects that hurt unambiguity and completeness; simulation is used for detecting defects that hurt consistency.

Sinnig et al. (2009) propose a syntax definition that formalizes the sequencing of use case steps and their types; based on these syntax a formal semantics based on *Labeled Transition System* (LTS) (Keller, 1976) 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.

The authors developed the **Use Case Analyzer tool** to automatically detect livelocks. They also propose a method for verifying refinement of use case models, namely checking their equivalence and deterministic reduction. Most of the checks focus on global properties of use case models, and only **sequential relationships** (precedence) among use cases are considered (See Table **67** in Appendix 2).

Zhao and Duan (2009) propose an approach to formalize use cases semantics with *Timed and Controlled Petri-Nets* (TCPN). A semi-structured natural language is proposed for use case syntax. The events in use cases can be sequential, conditional, iteration or concurrent (parallelism). Petri-Nets are

derived mapping use case events into sub Petri-Nets and linking them. Based on the obtained Petri-Net model, criteria to detect incompleteness, inconsistency and incorrectness properties are described (See Table 68 in Appendix 2). This approach evaluates properties of use cases and their associated use cases, separately; thus, **relationships** among use cases are no considered.

Somé (2010) proposes an approach for formalizing textual use cases via reactive Petri-Nets (Eshuis and Dehnert, 2003). They provided an algorithm for the generation of a reactive Petri-Net from textual use cases described using a formal syntax, and taking into account *include* and *extend* UML relationships and sequencing constraints using pre/post-conditions. The constructed reactive Petri-Net can be used for synthesis or analysis of **Consistency** properties defined in the transformation are satisfied (Table 69 in Appendix 2). This approach deals with sequential UML **relationships** among use cases (**include** and **extend**). However, the language to describe use cases does not deal with communication between concurrent use cases and other type of relationships among use cases.

2.5.3. Analysis Approaches Compared

The studies reported in the literature for analysis of **requirement statements** are focused mainly in analysis of **static aspects** of individual requirement sentences in order to detect and correct ambiguity issues. The evaluation of the different techniques were based on the following aspects: What **requirement representation** is used?, What **analysis technique** is used?, Are **relationships among requirements** considered for analysis?, Is it **tool-supported**?, Does it detect **unambiguity** defects?, Does it detect **completeness** defects?, Does it detect **consistency** defects?, Does it detect **correctness** defects?, Is it **applicable to scenario** representations?. Table 4 summarizes the results in a matrix.

Most of the reported techniques (Section 2.5.2.1.1) support the detection of ambiguity defects based on indicators databases and NLP techniques (except Wilson et al., 1997). Only Lucassen et al. (2015) consider the relationships among requirements for analysis. Lucassen et al. (2015) and Arora et al. (2015) check completeness in relation to conformance to requirements templates; Gnesi

et al. (2005) and Lucassen et al. (2015) supports the consistency checks by making explicit related requirements.

Most of these techniques are applicable to analysis of requirements described as scenario representations, since analysis of ambiguity can be applied to individual scenario elements or steps.

Table 4 – Comparing Requirement Statements Static Analysis Techniques

	Wilson et al., 1997	**Gnesi et al., 2005**	**Tjong, 2008**	**Femmer et al., 2014**	**Arora et al., 2015**	**Lucassen et al., 2015**
Requirement Representation	No	No	No	No	Rupp's; EARS;	User Story;
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;
Relationships Among Requirements	No	No	No	No	No	Yes
Tool-supported	Yes	Yes	Yes	Yes	Yes	Yes
Unambiguity	Yes	Yes	Yes	Yes	Yes	Yes
Completeness	No	No	Partial	Partial	Yes	Yes
Consistency	No	Partial	No	No	No	Partial
Correctness	No	No	No	No	No	No
Applicable to Scenario	Yes	Yes	Yes	Yes	Yes	Partial

Most of the studies reported in the literature for analysis of **scenarios** are focused mainly in analysis of **static aspects** of scenario elements. The evaluation of the different static analysis techniques were based on the following aspects: What **scenario representation** is used?, Is there a **syntax for scenario**?, What **analysis technique** is used?, Are **relationships among internal components of scenarios** considered (e.g. actors, steps, extensions) for analysis?, Are **relationships among scenarios** considered for analysis?, Is it **tool-supported**?, Does it detect **unambiguity** defects?, Does it detect **completeness** defects?, Does it detect **consistency** defects?, Does it detect **correctness** defects?. Table 5 summarizes the results in a matrix.

Most of the reported techniques support the detection of ambiguity indicators and completeness defects by applying inspection checklists; Ciemniewska and Jurkiewicz (2007), Sinha et al. (2010) and Liu et al. (2014) also take advantage of NLP techniques. Phalp et al. (2007) does not consider the relationships among scenarios for analysis. Only Ciemniewska and Jurkiewicz (2007), Sinha et al. (2010) and Liu et al. (2014) present tool-supported techniques.

Most of these techniques check completeness in relation to conformance to scenario templates. Leite et al. (2000) provides heuristics for consistency checking; others support partially the consistency checking, e.g., Ciemniewska

and Jurkiewicz (2007) provide heuristics for automatic detection of use case duplication. Only Leite et al. (2000) checks the consistency between scenarios and domain information (LEL).

Most of these techniques do not provide support for analysis of dynamic aspects of scenarios, because they do not provide execution semantics or insights for mapping into executable models.

Table 5 - Comparing Scenarios Static Analysis Techniques

	Leite et al., 2000	**Anda and Sjoberg, 2002**	**Ciemniewska** and Jurkiewicz**, 2007**	**Phalp et al., 2007**	**Sinha et al., 2010**	**Liu et al., 2014**
Scenario Representation	Scenario	Use Case Diagram; Use Case;	Use Case	Use Case;	Use Case diagram; Use Case;	Use Case;
Syntax for Scenarios	Yes	Partial	Partial	Yes	Yes	Partial
Analysis Technique	Checklist; Heuristics	Checklist	Heuristics; NLP;	Checklist	Checklist; NLP;	Checklist; NLP;
Relationships Among Internal Components	Yes	Partial	Yes	Partial	Yes	Partial
Relationships among Scenarios	Yes	Partial	Partial	No	Partial	Partial
Tool-supported	Manual	Manual	Yes	Manual	Yes	Yes
Unambiguity	Partial	Partial	Yes	Partial	Partial	Partial
Completeness	Yes	Yes	Yes	Yes	Yes	Partial
Consistency	Yes	Partial	Partial	Partial	Partial	Partial
Correctness	Partial	Partial	Partial	Partial	Partial	No

Most of the studies reported in the literature for analysis of **dynamic aspects** of **scenarios** are focused on translating natural language-based scenarios into formal representations, and take advantage of execution semantics of formal languages to simulate the behavior of set of scenarios and detect defects from possible interactions. The evaluation of the different dynamic analysis techniques were based on the following aspects: What **scenario representation** is used?, Is there a **syntax for scenario**?, What **analysis technique** is used?, Are **relationships among internal components of scenarios** considered (e.g. actors, steps, extensions)?, Are **relationships among scenarios** considered for analysis?, Are **non-explicit relationships among scenarios** considered for analysis?, Are related scenarios **integrated for whole 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?. Table 6 summarizes the results in a matrix.

Most of the reported techniques support partially the detection of unambiguity and completeness defects by applying inspection checklists; these

checklists mainly evaluate the conformance to previous syntax defined. Lee et al. (2001), and Zhao and Duan (2009) do not consider the relationships among scenarios for analysis. Only Lee et al. (1998) take into account non-explicit relationships among scenarios for analysis. Most of the techniques that consider relationships among scenarios integrate the related scenarios for a whole analysis. Lee et al. (1998) and Glinz (2000) do not present tools to support the analysis.

Most of these techniques are based on mapping to formal representations to take advantage of execution semantics, such as Petri-Nets, LTS or Statecharts; from these representations, a reachability graph is generated for analysis of the behavior of a set of scenarios and their relationships.

Most of these techniques do not take into account the state explosion issue of the generated reachability graph. Only Lee et al. (1998) manages the state explosion issue of reachability analysis by the use of **slices** strategy.

Petri-Net based techniques represent the interaction among concurrent scenarios in a more intuitive way, by fusing places or transitions to show the communication among concurrent scenarios.

Most of these techniques check consistency and correctness using equivalent formal representations, and they rarely map the results of the analysis to scenarios or the scenario relationships.

Table 6 - Comparing Requirement Statements Dynamic Analysis Techniques

	Lee et al., 1998	**Lee et al., 2001**	**Denger et al., 2005**	**Zhao and Duan, 2009**	**Sinnig et al., 2009**	**Somé, 2010**
Scenario Representation	Use Case; Action-Condition table;	Use case; Sequence Diagram;	Use Case;	Use Case;	Use Case; Use Case diagram;	Use Case; Use Case diagram;
Syntax for Scenarios	No	Partial	Yes	Yes	Yes	Yes
Analysis Technique	Constraints-based Modular Petri-Net;	Time Petri-Nets;	Checklist; Statechart;	Timed and Controlled Petri-Nets;	LTS;	Reactive Petri-Net;
Relationships Among Internal Components	No	Yes	Partial	Partial	Yes	No
Relationships among Scenarios	Yes	No	Partial	No	Yes	Partial
Non-explicit Relationships among Scenarios	Yes	No	No	No	No	No
Integration of Related Scenarios for Whole Analysis	Yes	No	Partial	No	Partial	No
Tool-supported	No	Partial	Partial	Partial	Partial	Yes
State Explosion Management	Slices	No	No	No	No	No
Unambiguity	Partial	Partial	Partial	Partial	Partial	Partial
Completeness	Yes	Partial	Partial	Partial	Partial	Partial
Consistency	Yes	Partial	Partial	Partial	Partial	Partial
Correctness	Yes	Partial	Partial	Partial	Partial	Partial

2.5.4. Research Gaps

Due to the focus of our research is the analysis of **requirements** described as **scenario** representations, we have identified the following gaps in scenario-based analysis techniques:

- We have not seen an explicit understanding of how the main quality characteristics are related to each one, i.e. they have positive and negative contributions among them. Only the **requirement statements** analysis works decomposed the main qualities (Unambiguity) in related properties (Categories of indicators) and modeled their impacts.
- Most of the techniques for static or dynamic analysis of scenarios do not take into consideration the results achieved by **requirement statements** analysis techniques in finding ambiguity indicators. Mainly, the works of Gnesi et al. (2005), Tjong (2008) and Lucassen et al. (2015) demonstrate that NLP techniques may be effective in detecting ambiguity, in some cases with 100% recall and high precision.
- Most of the analysis techniques are applied on scenarios written using formal syntax rules, or based on purely textual descriptions (conform to the template of Cockburn, 2001). Therefore, there is a lack of systematic procedures on how to represent scenarios. In case of techniques based on Cockburn (2001), intermediate models are created for mapping into formal representations.
- Only few of the works, like Somé (2010) apply consistency rules for verifying the preservation of the consistency between scenarios and their equivalent formal representations.
- Scenarios are rarely independent; they interact, in some cases non-explicitly. Most of the existing proposals only consider sequential relationships (e.g. extend or include UML relationships) among scenarios; they do not propose constructs or heuristics to identify non-explicit relationships. Non-explicit relationships can hide non-sequential interactions (indistinct sequential order, concurrency or parallelism) among scenarios.

- Most of the existing techniques do not describe systematic procedures on how integrate the equivalent formal representations of a set of related scenarios (they interact) into a whole representation, in order to evaluate the behavior and detect defects from the relationships in a set of scenarios. Only Lee et al. (1998) present a formal approach to integrate formal use cases with sequential and non-sequential (concurrent) relationships. Sinnig et al. (2009) and Somé (2010) give some details for the integration of sequentially related scenarios.
- The state explosion issue is a big problem when reachability graphs are generated for analysis of complex systems. Only Lee et al. (1998) propose a strategy to lead with this problem.
- When equivalent formal representation are evaluated using formal analysis strategies, and errors are found, the results of the analysis must be described in a comprehensive way, and mapped to defects within scenarios or the scenarios' relationships. None of the existing approaches return a feedback in a comprehensive way and tracing the errors from formal representations to scenarios.

3
A Quality Model for Scenarios

This section begins with a general introduction of Software Requirements Specification (SRS) quality and the importance of scenario-based SRS (Section 3.1). Next, In Section 3.2 we employ the non-functional requirements (NFR) approach (Chung et al., 2000) to model the relationships between unambiguity, completeness, consistency and correctness qualities. This section also shows how a quality property can be evaluated by searching defect indicators. Finally, Section 3.3 discusses the benefits of the proposed Quality Model for Scenarios.

3.1.
Quality in Scenario-based SRS

Many problems and high-risk issues that arise during the software development process are related with deficiencies at RE activities. Assessing that SRS satisfices (relative satisfaction) the necessary quality is crucial to the success of any software development project, since the SRS is the anchor for software development. However, assessing the quality of a SRS is not a simple process, mainly, because a system must often support multiple stakeholders with different viewpoints and needs, which may be contradictory.

Nowadays, Scenario-based representations are frequently used in RE for requirements specification. A scenario-based SRS has the potential to influence positively the SD process, requirements engineers need to pay special attention to its quality, in special with respect to ***unambiguity***, ***completeness***, ***consistency*** and ***correctness*** since they will anchor further development.

There are several different templates or syntax for writing scenarios, and most of common components used to detail scenarios are:

- **Title**/Name: Name that identifies the scenario;
- **Goal**: Purpose of the scenario;
- **Pre-condition**: System state before the scenario can start;
- **Post-condition**: System state after the scenario is performed;

- **Actors**: Persons, device or organization structures (active entities) that have a role in the scenario;
- **Episodes**/Main Flow: Steps to achieve the goal of the scenario;
- **Exceptions**/Alternative Flows: Exceptions from the course of events;

Other components that might be useful to detail in scenario descriptions are: **Resources** (Leite et al., 2000; Sinha et al., 2010) and **Context** (Leite et al., 2000).

As stated before (Chapter 2), scenarios are usually written in natural language, however, natural language is by definition *ambiguous* leading to *incomplete*, *inconsistent* and *incorrect* SRS. *Ambiguity* occurs when two or more users have different interpretations of the same requirement. *Incomplete* requirements occur because the world is complex; as such, users or clients are not able to identify and develop all relevant requirements. *Inconsistent* requirements occur when two or more users have conflicting requirements, or the captured requirements are internally inconsistent when one or more requirements override others. *Incorrect* requirements may occur when the acquired requirements do not accurately reflect the facts, or erroneous predicts about future states.

Numerous techniques have been developed to deal with these quality problems in SRS and each one with its own context of applicability. However, most SRS still do not meet these qualities. According to Glinz (2000), it is not only a problem of applying the right methods and processes for SRS until they yield the desired qualities; the qualities themselves are part of the problem.

So, in order to understand these qualities it is necessary to model the relationships between *unambiguity*, *completeness*, *consistency* and *correctness*.

3.2. Modeling Correctness as Non-functional Requirements

Our proposal is one of the first to represent *unambiguity*, *completeness*, *consistency* and *correctness* as non-functional requirements based on NFR framework (Chung et al., 2000). Moreover, our contribution also exposes the links and impacts between the properties related to the main NFRs. We introduce a novel perception of *correctness* and its complex relationships with *unambiguity*, *completeness* and *consistency* describing it as a quality that should be satisficed by contributions of related qualities or properties.

Based on the literature, we have developed a Soft-goal Interdependency Graph (SIG) for SRS *Correctness* (Figure 15), which will be the base for cataloged information and will be detailed by a series of decomposition or contribution interdependencies. In order to elaborate the SIG to achieve Correctness, we defined a set of two steps, which are detailed below.

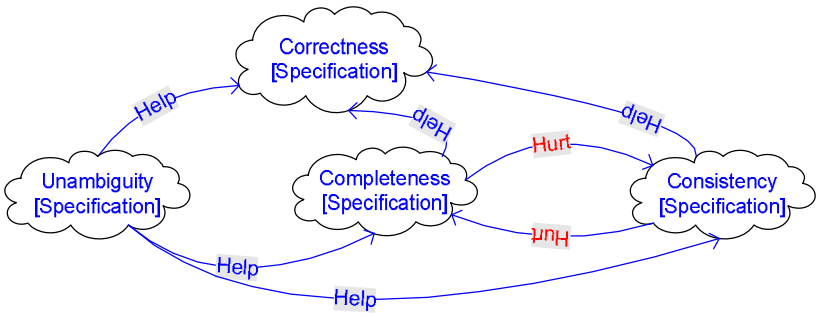


Figure 15 – Initial SIG of SRS Correctness.

3.2.1. Defining the Main NFRs

In the NFR Approach, *unambiguity*, *completeness*, *consistency* and *correctness* are non-functional requirements that need to be satisficed (relative satisfaction) by the specifications.

In our work, it is assumed that: (1) *correctness* is the most important quality; and (2) there is an important causal relationship between *unambiguity*, *completeness*, *consistency* and *correctness* of SRS. Glinz (2000) uses the term *adequacy* instead of *correctness*, and focuses on *adequacy* as the most important quality.

Zowghi and Gervasi (2003) argue as increasing the completeness of a SRS can decrease its consistency and hence affect the correctness of the final product. Conversely, improving the consistency of the SRS can reduce the completeness, thereby again diminishing correctness. It is frequently the case that in an attempt to maintain consistency within the requirements we remove one or more requirements from the specification and fail to preserve its completeness. Conversely, when we add new requirements to the specification to make it more complete, it is possible to introduce inconsistency in the specification (Zowghi and Gervasi, 2003).

In the context of scenario-based SRS, *unambiguity* concerns a sentence (scenario consists of many sentences); *completeness* concerns a single scenario

and its internal components, and a set of scenarios and their relationships; *consistency* concerns a set of scenarios and their relationships.

3.2.1.1. Unambiguity

A specification is ***unambiguous*** if and only if, every requirement stated therein has only one interpretation (IEEE, 1998). Although, *unambiguity* is very difficult to achieve and evaluate because most of the related indicators are subjective; there are defect indicators that can be found. These indicators provide evidence against the existence of *unambiguity*. These indicators can be grouped and categorized in properties that contribute negatively to ***Unambiguity***: *Vagueness*, *Subjectiveness*, *Optionality*, *Weakness*, *Multiplicity*, *Implicitly* and *Quantifiability*. Other properties that contribute positively to unambiguity are: *Readabiity* and *Minimality*.

Due to inherent *ambiguity* in natural language-based scenario specifications; NLP techniques can be useful to address several problems that impacts negatively to *unambiguity* by evaluating linguistic aspects of internal scenario components (Title, Goal, Episodes/Steps, Conditions, Exceptions, and so on). Most of these problems are defects that contribute to *unambiguity* properties, and they can be identified by performing a lexical or syntactical analysis.

Lexical analysis can be performed to search defect indicators that contribute to *Vagueness*, *Subjectiveness*, *Optionality, Multiplicity, Quantifiability, Readabiity* and *Minimality*. This analysis is based on the occurrence of special *terms* (words or phrases) in the sentences or the number of elements in the sentences (Gnesi et al., 2005).

Syntactical analysis is necessary to detect defects related to *Weakness and Implicitly* properties; however, lexical parsers can be also used to search *ambiguity* indicators, e.g. weak phrases (Wilson et al., 1998; Tjong, 2008).

Based on results achieved by requirement statements analysis techniques (Wilson et al., 1998; Gnesi et al., 2005; Tjong, 2008), we organized the set of properties that contribute to ***unambiguity*** in NL-based scenarios, which can be evaluated by following a checklist with verification heuristics to search defect indicators. The indicators of these properties are collected into *dictionaries* that contain *frequently used words* or *phrases* characterizing defects, and evaluated by

metrics which can be easily identified and counted by computer programs (See Table 7).

Table 7 – Properties Related to Unambiguity.

Property	Description	Heuristic	Indicator
Vagueness	The sentence contains words or phrases having a non-uniquely quantifiable meaning (Gnesi et al., 2005).	Check that a sentence does not contain vague terms	A sentence contains: *adaptability, additionally, adequate, aggregate, also, ancillary, arbitrary*, …
Subjectiveness	The sentence contains words or phrases expressing personal opinions or feeling (Gnesi et al., 2005).	Check that a sentence does not contain subjective words	A sentence contains: *similar, better, similarly, worse, having in mind, take into account, take into consideration, as possible.*
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	A sentence contains: *as desired, at last, either, eventually, if appropriate*, ...
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	A sentence contains: *can, could, may, might, ought to, preferred, should, will, would.*
Multiplicity	The sentence has more than one main verb, subject or object.	Check that a sentence does not contain conjunction or disjunction words	A sentence contains: *and, or, and/or*
Implicitly	The sentence does not specify the subject or object by means of its specific name but uses pronoun or other indirect reference (Gnesi et al., 2005).	Check that a sentence does not contain implicit words	A sentence contains: *anyone, anybody, anything, everyone, he, her, hers, herself*, ...
Quantifiability	Terms used for quantification can lead to ambiguity if not used properly (Arora et al., 2015).	Check that a sentence does not contain quantification words	A sentence contains: *all, any, few, little, many, much, several, some*.
Readabiity	It measures how easily an adult can read and understand the sentence or Document (Wilson et al., 1997).	This metric is the Coleman-Liau Formula readability metric: (5.89*chars/words-0.3*sentences/(100*words)-15.8]).	The reference value of this formula for an easy-to-read technical document is 27.60, if it is < 17.10 and > 55.80 the document is difficult-to-read.
Minimality	A sentence contains nothing more than basic attributes (Lucassen et al., 2015).	Check that a sentence does not contain additional information	A sentence contains a Text after a: *dot*, *hyphen*, *semicolon* or other punctuation mark.

3.2.1.2. Completeness

A specification is ***complete*** if all relevant requirements are present and each requirement is fully developed (Boehm, 1979). *Incomplete* requirements occur because the world is complex; as such, users or clients are not able to fully understand the impact of present decisions.

Although, ***completeness*** is very difficult to define and evaluate because it also depends on external aspects, and violations are hard to detect; there are internal aspects that can be measured. In Leite et al. (2000), the evaluation of completeness in scenario-based specifications is done following a checklist with verification heuristics to detect violations of properties in internal elements of scenarios and their relationships.

Based on Leite et al. (2000), we re-organized the properties that contribute positively to ***completeness***. We understand that a fully developed SRS presents properties related to internal aspects of scenarios (intra-scenario) and their relationships (inter-scenario). The intra-scenario properties include: ***Atomicity***, ***Simplicity***, ***Uniformity***, ***Usefulness*** and ***Conceptually Soundness***. The inter-scenario properties include: ***Integrity***, ***Coherency*** and ***Uniqueness***. Other important property related to ***completeness*** is ***Feasibility***. For each property, we defined verification heuristics for searching defect indicators that hurt the property.

Table 8, Table 9, Table 10 and Table 11 show the properties that contribute to completeness and heuristics to search common defect indicators that contribute negatively to them. These verification heuristics are driven by syntax checks and by cross-referencing the related scenarios. Some of these heuristics were learned or reported by related work during their experience with scenarios or use cases analysis.

Table 8 – Intra-scenario Properties Related to Completeness (Continued on Table 9).

Property	Description	Heuristic	Indicator
Atomicity	A scenario expresses exactly one situation (Adapted from Lucassen et al., 2015).	1. Check that Title defines exactly one situation;	*and, or, and/or*;
		2. Check that Goal satisfies exactly one purpose;	*and, or, and/or*;
		3. Check that Title contains a verb in infinitive (base) form and an object;	Missing *Action-Verb* in Title; Missing *Object* in Title;
Simplicity	A scenario should be as readable as possible;	1. Check that each Episode/Exception consists of a subject, a verb, and optionally, an object and a prepositional phrase (It is not a complex sentence, Ciemniewska and Jurkiewicz, 2007);	*Episode/Exception* contains more than one *Action-Verb*; *Episode/Exception* contains more than one *Subject*; Missing *Subject*; Missing *Object*;
		2. Check that Episode/Exception is described from user point of view, i.e., the present simple tense and active form of a verb should be used (Ciemniewska and Jurkiewicz, 2007);	The *Action-verb* is not in the *third form*;
		3. Check that Title does not contain extra unnecessary information (Adapter from Phalp et al., 2007).	*Title* contains text between brackets (e.g. (...), {...}), URLs, HTML
		4. Check that Episode coincidence only takes place in different situations;	Duplicated *Episode Id* or *sentence*;
		5. Check that 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);	More than one *Episode* inside a nested *IF*;
		6. Check that Exception is handed by a simple action, i.e., if the interruption causes the execution of a sequence of sentences, then this sequence should be extracted to a separate scenario (Ciemniewska and Jurkiewicz, 2007);	More than one *Sentence* inside a *Exception Solution*;
Uniformity	Each scenario element is constructed using defined scenario model.	1. Check the completeness of each scenario element (Leite et al., 2000);	Missing *Title* Missing *Goal* Missing *Actors* Missing *Resources* *Context* does not contain its relevant sub-components Missing *Episodes* *Episode* does not contain its relevant parts (Id, Sentence) *Exception* does not contain its relevant parts (Id, Cause, Solution)

Table 9 – Intra-scenario Properties Related to Completeness.

Property	Description	Heuristic	Indicator
Usefulness	A scenario does not contain superfluous information, i.e., there should be consistency among scenario elements. (Adapted from Anda et al., 2009).	1. Check that every Actor participates in at least one episode;	Actor does *not participate* in the situation;
		2. Check that every Actor mentioned in episodes is included in the Actor element;	Missing *Actor* in Actors element;
		3. Check that every Resource is used in at least one episode;	Resource that is *not used* in the situation;
		4. Check that every Resource mentioned in episodes is included in the Resource element;	Missing *Resource* in Resources element;
		5. Ensure that step numbering between the main flow and alternative/exception flow are consistent (Liu et al., 2015);	Branching *Episode* of an exception is *missing*;
		6. Check the existence of more than two and less to 10 episodes per scenario (Leite et al., 2000; Ciemniewska and Jurkiewicz, 2007);	Number of *episodes* in each scenario is less than 3 or more than *9*;
Conceptually Soundness	Internal scenario elements are semantically coherent, i.e., scenario elements satisfy the scenario goal (Leite et al., 2000).	1. Check that the Title describes the Goal;	The corresponding *verbs* and *objects* appearing in the two compared sentences are not the same
		2. Ensure that the set of Episodes satisfies the Goal and is within the Context;	*Difficult to be measured by an automatic tool;*
		3. Ensure that actions presents in the Pre-conditions are already performed;	
		4. Ensure that Episodes contain only action to be performed;	Missing *Action-Verb* in episode sentences;
		5. Ensure that Episode condition contains Linking-Verbs;	Missing *Linking-Verb* in episode conditions;
		6. Ensure that Pre-conditions contain State-Verbs;	Missing *State-Verb* in Pre-conditions;
		7. Ensure that Post-conditions contain State-Verbs;	Missing *State-Verb* in Post-conditions;
		8. Ensure that Exception solution contains only action to be performed;	Missing *Action-Verb* in exception solution;
		9. Ensure that Exception cause contains Linking-Verbs or State-Verbs;	Missing *Linking-Verb or State-Verb* in exception causes;

In Table 10, ***Integrity*** and ***Coherency*** properties are evaluated checking the main scenario against the related scenarios to it. ***Uniqueness*** properties are evaluated checking the main scenario against the other scenarios (related or not).

Table 10 – Inter-scenario Properties Related to Completeness.

Property	Description	Heuristic	Indicator
Integrity	Whenever a scenario includes an explicit relationship on another scenario, the related scenario should exist as another scenario within the set of scenarios.	1. Check that every included scenario exists within the set of scenarios (Leite et al., 2000);	*Pre-condition* identified as related scenario *does not exist* within the set of scenarios; *Post-condition* identified as related scenario *does not exist* within the set of scenarios; *Episode sentence* identified as related scenario *does not exist* within the set of scenarios; *Exception solution* identified as related scenario *does not exist* within the set of scenarios; *Constraint* identified as related scenario *does not exist* within the set of scenarios;
		2. Check that every Exception is treated by a scenario (Leite et al., 2000);	Complex *Exception Solution* must be treated by a scenario;
		3. Check that a Pre-condition (not described as another scenario) of a scenario is satisfied by a Post-condition of other scenario, i.e., it is possible to infer relationships from *pre-condition/post-condition* (Leite et al., 2000);	Missing *pre-condition/post-condition;*
Coherency	Internal elements of explicitly related scenarios should be precise and use a common terminology, e.g. pre-conditions of sub-scenarios are coherent with main scenario pre-conditions.	1. Check coherence between the related scenario Pre-conditions and the main scenario Pre-conditions;	*Difficult to be measured by an automatic tool;*
		2. Check that Geographical and Temporal location of the related scenarios are equal or more restricted than those of the main scenario (Leite et al., 2000);	Related scenario Geographical location is not in the set of Geographical locations of root scenario; Related scenario Temporal location is not in the set of Temporal locations of root scenario;
		3. Check that every referenced scenario does not reference the main scenario (Adapted from Sinnig et al., 2009);	Circular inclusion between two scenarios;
Uniqueness	A scenario is unique when no other scenario is the same or too similar, i.e., duplicates are avoided because they are source of inconsistencies (Adapted from Lucassen et al., 2015);	1. Check that the Title of a scenario is not already included in another scenario;	*Title* coincidence between two scenarios;
		2. Check that the Goal of a scenario is not already included in another scenario;	*Goal* coincidence between two scenarios;
		3. Check that the Context Pre-condition of a scenario is not already included in another scenario;	*Pre-condition* coincidence between two scenarios;
		4. Check that the set of Episodes of a scenario is not already included in another scenario;	*Episodes* coincidence between two scenarios;
		5. Check the similarity of the scenario with other scenarios using syntactic analysis;	Titles share the same *Action-Verb* and the direct *Object*;

Table 11 – Feasibility Property Related to Completeness.

Property	Description	Heuristic	Indicator
Feasibility	It is possible to perform each operation described in a scenario and each internal/external condition is not violated.	1. Check that is possible to *derive an initial system design from the current scenario* (Adapted from Denger et al., 2005);	There are not relationships among scenarios;
		2. Check that *initial system design* does not contain *isolated sub-systems*;	Unreachable operations;

3.2.1.3. Consistency

A specification is ***consistent*** when two or more requirements are not in conflict with one another or with governing specifications and objectives (Boehm, 1979). *Inconsistent* requirements occur when two or more users have conflicting requirements, or the captured requirements are internally inconsistent when one or more requirements override others.

Evaluation of ***consistency*** with respect to external specifications is very difficult to perform. Consistency defects are difficult to detect or only with much-effort. However, internal aspects of consistency can be evaluated when the behavior of a set of scenarios is simulated, and defects are identified in scenario relationships.

One of the main strategies to ensure the ***consistency*** is the evaluation of dynamic aspects of a SRS. This is done by first mapping scenario representations into executable models, and performing a rigorous behavioral analysis to detect violations (defects) of properties that contribute positively to ***consistency***. There exist several tools to perform rigorous analysis on executable models (e.g. Petri-Net). These tools generate a reachability graph which contains the different states of execution, and traverse this graph for searching defect indicators.

Table 12 shows the properties that contribute to ***consistency*** and verification heuristics for searching defect indicators that hurt these properties. Dynamic properties that influence the ***consistency*** are: ***Non-interferential***, ***Boundedness***, ***Reversibility*** and ***Liveness***.

Table 12 –Properties Related to Consistency.

Property	Description	Heuristic	Indicator
Non-interferential	Every operation that negatively affect on others should be identified.	Check Non-determinism: A non-deterministic behavior occurs when a set of operations are simultaneously enabled. If the reachability graph reveals non-deterministic execution paths, a *warning* is reported to indicate wrong information (Lee et al., 1998; Lee et al., 2001).	Simultaneously enabled operations;
Boundedness	This property refers to the limited capacity of a communication channel or shared resource.	Check Overflow: An executable model is overflowed when the number of elements in some communication channel or resource exceeds a finite capacity (Zhao and Duan, 2009).	Overflowed resource;
Reversibility	The behavior should reach its initial state again.	Check Reversibility: Reversibility of an executable model guarantees that the described behavior reaches its initial state again. If the executable model is not reversible, the automatic error recovery is not possible (Cheung et al., 2006).	There are no a path from an operation to the initial state;
Liveness	Every operation can be executed in the future.	Check Liveness: Liveness is closely related to the complete absence of deadlocks. An executable model is *deadlocked* if no process can make any progress, generally because each is waiting for communication with others (Lee et al., 1998).	Path to deadlock; Never enabled operations;

3.2.1.4. Correctness

A specification is ***correct*** if, and only if, every requirement stated therein is one that the software shall meet (IEEE, 1998). *Incorrect* requirements may occur when the acquired requirements do not accurately reflect the facts, or erroneous predicts about future states.

Correctness is the main requirements quality, and it is difficult to evaluate and achieve. Therefore, having an *unambiguous, complete* and *consistent* set of scenarios contributes positively to more *correct* SRS.

3.2.2. Modeling the SIG

In order to evaluate *unambiguity, completeness, consistency* and consequently *correctness*, we apply the NFR qualitative reasoning approach (Chung et al., 2000); the goal here is to achieve good *correctness* in scenario-based SRS.

Figure 16 illustrates the SIG that models the SRS *Correctness* and how SRS *Unambiguity*, *Completeness* and *Consistency* impact positively (help) to *Correctness*. We assume that a SRS is more *correct*, if it is perceived as *unambiguous*, *complete* and *consistent* with respect to real user's needs. Interdependency between SRS Consistency and Completeness impacts negatively (hurt) on both (Zowghi and Gervasi, 2003).

Properties that contribute negatively to *Unambiguity* are modeled using HURT links – in *Vagueness*, *Subjectiveness*, *Optionality*, *Weakness*, *Multiplicity*, *Implicitly* and *Quantifiability*. Also, properties that contribute positively to *Unambiguity* are modeled using HELP links – in *Readabiity* and *Minimality* soft-goals.

Completeness and *Consistency* are decomposed – using AND links – in *Internal* and *External* soft-goals, following the lead of (Boehm, 1979) and (Zowghi and Gervasi, 2003). Evaluating External Completeness and External Consistency is a hard problem because it depends on external specifications, external domain models and user's needs satisfaction.

Properties that contribute positively to *Internal Completeness* are modeled using HELP links – in *Atomicity, Simplicity, Uniformity, Usefulness, Conceptually Soundness*, *Integrity*, *Coherency, Uniqueness* and *Feasibility* soft-goals. These soft-goals can be operationalized by: (1) Writing Scenarios using Regular Languages (Hsia et al., 1994; Cheung et al., 2006), OR (2) Writing Scenarios following concrete syntax rules (Leite et al., 2000; Anda and Sjoberg, 2002; Denger et al., 2005; Phalp et al., 2007; Sinha et al., 2010), OR (3) Analyzing scenarios using NLP techniques.

Properties that contribute positively to *Internal Consistency* are modeled using HELP links – in *Non-interferential*, *Boundedness*, *Reversibility* and *Liveness* soft-goals. These soft-goals can be operationalized by: (1) Writing scenarios using Regular Languages (Hsia et al., 1994; Cheung et al., 2006), OR (2) Analysis of Scenarios with Petri-Nets (Lee et al., 1998; Lee et al., 2001; Zhao and Duan, 2009; Somé, 2010); OR (3) OR Analysis of Scenarios with LTS (Sinnig et al., 2009).

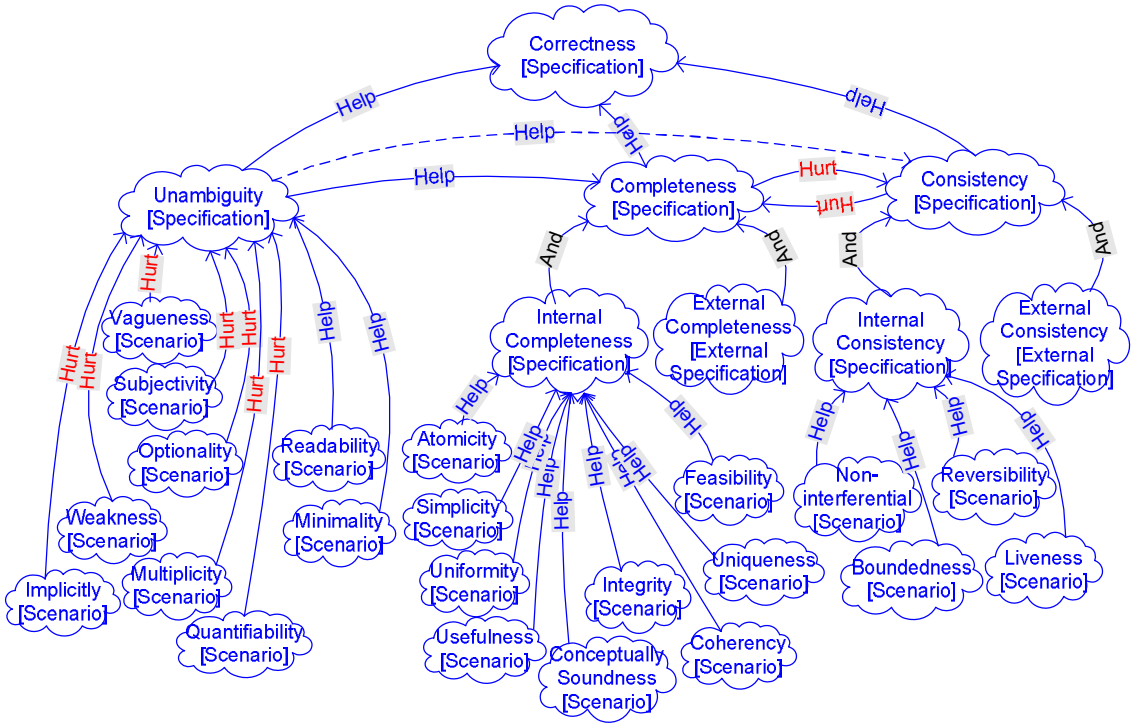


Figure 16 – SIG of SRS Correctness.

3.3. Final Considerations

The process of scenario-based SRS verification is a complicated activity; no single solution is effective to resolve the challenges of dealing with Unambiguity, Completeness, Consistency and Correctness. And, there is a lack of systematic approaches to model and organize the related properties to Unambiguity, Completeness, Consistency and Correctness of scenario-based SRS.

We have used the NFR framework to help the organization (Quality Model for Scenarios) of properties that contribute to Unambiguity, Completeness, Consistency and Correctness qualities with operationalizations using NLP techniques, Petri-Nets or LTSs, which can enable the automated SRS verification.

The quality model for scenarios can be used to evaluate static and dynamic properties of scenario-based SRS; and, it provides benefits due to the following reasons: (1) it identifies properties to be evaluated due to individual and interacting scenarios; (2) it shows what kind of operationalizations can be made to support the evaluation of properties; and (3) it is possible to reuse the patterns and specialize them for more specific scenario languages.

4
Scenario Analysis Approach

Scenarios are the main technique for modeling user requirements, which have been widely adopted by user-oriented approaches for software development. Due to natural language, defects are inevitably introduced in scenario descriptions. In this chapter, we discuss our approach for detecting defects in acquired *scenario descriptions*. It detects wrong information, missing information and erroneous situations that can be hidden within scenarios and their relationships with related scenarios.

For it, we (1) describe scenarios using a **Restricted-from of Natural Language (RNL) Scenario** technique, which presents a concrete grammar based on linguistic patterns to write sentences and describes the relationships among scenarios, and heuristics to identify non-explicit relationships; (2) instantiate the **Quality Model for Scenarios** (Chapter 3); and (3) consider the results achieved by NLP and Petri-Net based related work.

In our scenario analysis approach: ***First***, requirements engineers start to describe the different functionalities, services or situations of the system as textual *scenarios*. ***Second***, irrelevant information within scenario elements are removed. ***Third***, by an automatic transformation, an initial system design is derived by translating these *scenarios* into *Place/Transition Petri-Nets*, and synthesizing them into a consistent whole *Petri-Net* which represents the relationships among related scenarios. ***Fourth***, from these representations (Scenarios and their resulting Petri-Nets), defects that hurt *unambiguity* and *completeness* of scenarios are detected by analyzing structural properties of scenarios, and defects that hurt *consistency* and *correctness* of scenarios are detected by analyzing behavioral properties of equivalent Petri-Nets. ***Fifth,*** the analysis outcome is formatted and returned to the requirements engineers. ***Sixth,*** if defects are found, the analysis feedback is used to improve the scenario descriptions, since the identified defects and their causes can be traced to the scenarios. The approach also shows the source of errors detected in equivalent

Petri-Nets, i.e., Petri-Net analysis errors are traced into defects in scenarios or their relationships. With the feedback provided by the approach, the requirements engineer can improve the scenario descriptions and then the process starts again in pre-process activity until no defects are detected.

Below in Figure 17, we detail the activities of our analysis approach using the SADT language (Ross, 1977). The activities two to five (Pre-process, Derive, Analyze and Generate) are performed automatically by the C&L (2015) tool.

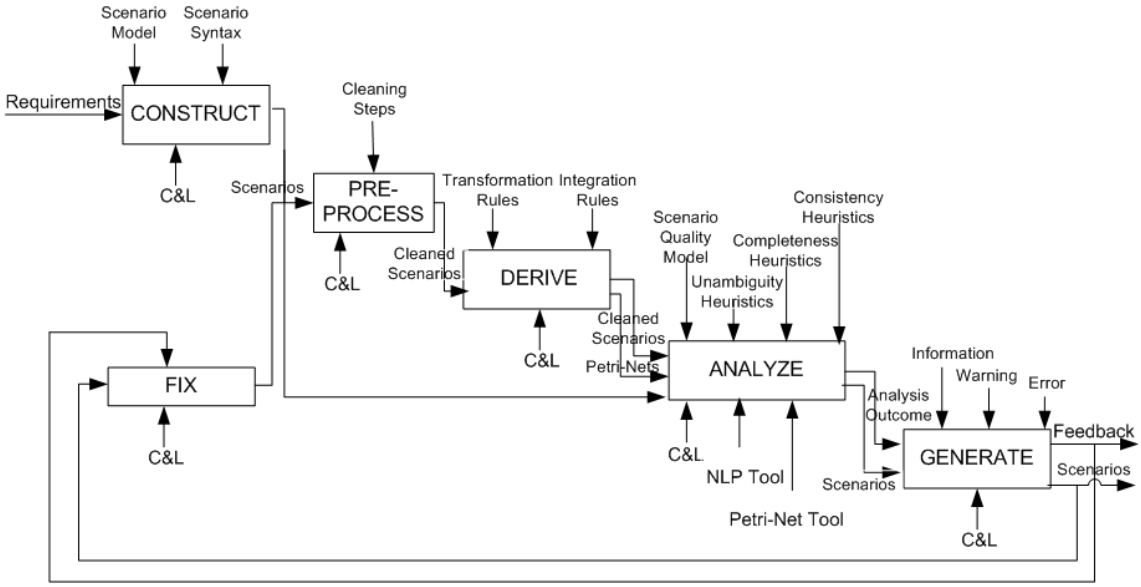


Figure 17– SADT of the Scenarios Analysis Approach.

4.1. Writing Restricted-form of Natural Language Scenarios

This activity is carried out by requirements engineers, which start to elicit the requirements and describe the different situations, functionalities, services or tasks of the system as scenario representations. In Leite et al. (2000), the scenario construction process is detailed and decomposed in other activities.

As mentioned before, the language used to write these scenarios is a Restricted-form of Natural Language (RNL). Using RNL it is possible to write imperative and declarative sentences. An imperative sentence describes actor events; and a declarative sentence describes actor or resource states. Thus, software requirements specifications can be described as clear and well-defined scenario descriptions.

The use of RNL restricts the vocabulary used to write scenarios and prevents the introduction of ambiguous sentences in the scenario specification, contributing to the quality of documentation. RNL is also necessary to define

syntax rules for sentences construction. Moreover, it helps the automatic transformation of textual scenarios into formal executable models.

The natural language based-scenario used in this work is an adaptation of a previous language (Leite et al., 2000). Unlike Leite et al. (2000), our focus is the analysis of scenarios. For this purpose, a new scenario language is defined by adding the *pre-condition* and *post-condition* attributes, and the *repetition structure control* to the grammar proposed in (Leite et al., 2000).

This sub-section begins with a definition of an abstract conceptual model for the proposed scenario language. Next, it presents a concrete grammar based on linguistic patterns to write sentences within scenarios using a restricted-form of natural language. It also describes the relationships among scenarios, and heuristics to identify non-explicit relationships.

4.1.1. Scenario

Scenario specifications capture system behaviors or situations in the domain (Leite et al., 2000) and, it helps the understanding of the requirements by the developers and other stakeholders.

In literature, the term *scenario* is used with different meanings in different contexts, and there is no clear distinction between scenarios and use cases. While some authors consider that each scenario corresponds to one use case (Glinz, 2000), others define a scenario as sequences of use case steps that represent different paths through a use case (Cockburn, 2001). According to Glinz (2000), a scenario may comprise a concrete sequence of interaction steps (*instance scenario*) or a set of possible interaction steps (*type scenario*). Based on definitions from Glinz (2000) and Leite et al. (2000), we stated a scenario definition that enables a further transformation.

A ***scenario*** is a collection of partially ordered *event occurrences*, each guarded by a set of *conditions* (pre-condition and post-condition) or restricted by constraints. An ***event*** is an operation or an interaction involving persons, organizations, system, environment, or system's components. A ***condition*** is an actor/resource/system state (e.g. the availability of some resource). An ***actor*** can be a user, organization, device, the system, system's components or agents in the environment; they have a role in the scenario or act on the Universe of Discourse.

Figure 18 presents an abstract conceptual model for the proposed scenario language, using a class diagram. According to our conceptual model, the ***scenario language*** is composed of the main entity Scenario, and the Context, Resource, Actor, Episode, Exception and Constraint entities.

In the proposed language, a ***scenario*** starts in an initial state (*context*) with all necessary *Resources* and *Actors*, and must satisfy a *Goal* that is reached by performing its *Episodes*. The *episodes* describe the operational behavior of the *situation*, which includes the main course of action and possible alternatives. An *Exception* can arise during the execution of *episodes*, and indicates that there exists an obstacle to satisfy the *goal*. The treatment to this exception does not need to satisfy the scenario *goal*. A scenario pre-condition, post-condition, constraint, episode or exception can be expressed by another scenario.

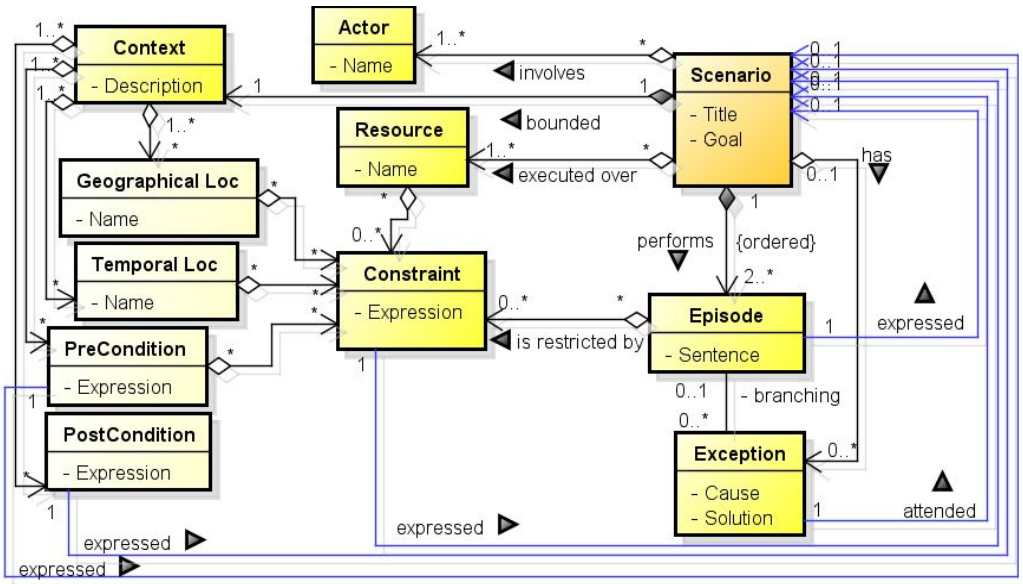


Figure 18 - Scenario Conceptual Model.

4.1.1.1. Title

The title of a ***scenario*** identifies the scenario using a declarative sentence, and it must be unique.

4.1.1.2. Goal

The goal describes the purpose of the scenario using a declarative sentence, and it gives a general idea about the scenario main purpose and how it is achieved.

4.1.1.3. Context

The context describes the scenario initial state using declarative sentences, and it must be described through at least one of the following sub-components: Pre-condition, Post-condition, Geographical or Temporal location.

Pre-condition expresses the initial state of the scenario. **Post-condition** expresses the final state of the scenario. **Geographical location** represents the physical set of the scenario. **Temporal location** is the time specification for the scenario development.

Pre-conditions, Post-conditions, Geographical locations and Temporal locations may be expressed by one or more simple sentences linked by the logical connectors ***AND*** or ***OR***. A pre-condition or post-condition sentence can be detailed in another scenario.

4.1.1.4. Resources

Resources are an enumeration of relevant physical elements or information (passive entities) that must be available in the scenario. They are used by the actors in the episodes to achieve scenario's goal. Resources must appear in at least one of the episodes.

4.1.1.5. Actors

Actors are an enumeration of persons, device or organization structures (active entities) that have a role in the scenario. They are directly involved with a *situation*. Actors must appear in at least one of the episodes.

4.1.1.6. Episodes

They are a set of actions that give an operational description of behavior. They represent the main flow, which is a sequence of steps where everything works as expected. An episode can be described as a scenario.

An ***episode e*** is a ***7-tuple (Id, Sentence, Type, Condition, Constraint, Pre-condition, Post-condition)***. Every *episode* is identified through an identifier *Id* and a *Type*. An episode performs an action - *Sentence* (imperative) that can use (or modify) resources and be executed by actors. Depending on the episode type,

conditions are added (conditional or loop episode). Optionally, the following attributes can be added: (1) *Constraints* that restrict the quality with witch the episode is performed; (2) *Pre-conditions* that we expect are already satisfied before the episode is performed; (3) *Post-conditions* that we expect will be achieved after the episode occurs. An episode is carried out only when the set of pre-conditions are satisfied.

Pre-condition and *post-condition* are described as declarative sentences involving relevant actor/resource/system states (e.g. the availability of a resource). They are different of context's pre-condition and context's post-condition because: (1) context's pre-conditions are the state of the system before the scenario is started; (2) context's post-conditions are the state of the system when the set of episodes are carried out. These attributes are important in the modeling, analysis and design synthesis of concurrent systems (Lee et al., 1998; Cheung et al., 2006). Pre-conditions and post-conditions can be expressed by one or more single sentences linked by the logical connectors **AND** or **OR**.

Episodes are simple, conditional, optional and loop ones. *Simple episodes* are those necessary to complete the scenario. *Conditional episodes* are those whose occurrence depends on internal or external conditions. *Optional episodes* are those that may or may not take place depending on conditions that cannot be detailed. *Loop episodes* can be used as repetition structures whose occurrence depends on internal or external conditions. Internal conditions may be due to alternative pre-conditions, actors or resources constraints and previous episodes. External conditions may be provided by external actors or another scenario. Conditions can be expressed by one or more single logical sentences linked by the logical connectors **AND** or **OR**.

Independently of its type, an episode can be expressed as a single action or can itself be conceived as a scenario, thus enabling the possibility of decomposition of a scenario in ***sub-scenarios***.

A sequence of episodes implies a precedence order, but a non-sequential order can be bounded by the symbol "#" allowing the grouping of two or more episodes. This is used to describe indistinct sequential order, concurrent or parallel episodes (#<Episode Series>#).

4.1.1.7. Exception

Exceptions are situations that prevent the proper course of the scenario. The treatment of the exception may be expressed through a sentence or detailed in another scenario. An exception hinders the achievement of the scenario goal, and it can describe an alternative or exceptional execution flow.

An ***exception ex*** is a ***4-tuple (Id, Cause, Solution, Post-condition)***. Every *exception* is identified through an identifier *Id*. An exception: (1) is caused by invalid input data or the lack or malfunction of a necessary resource (resource/system state) - *Cause*; (2) is treated by an imperative sentence – *Solution*; and optionally (3) may generate effects on the resource/system states, or simply produce a message – *Post-condition*.

An exception is always branched from an episode of the main execution flow. Causes and post-conditions can be expressed by one or more single sentences linked by the logical connectors **AND** or **OR**.

4.1.1.8. Constraint

Scope or non-functional requirement referring to a given entity, and described as a declarative sentence that restricts the quality with witch: (1) the *goal* is achieved, (2) a *resource* is needed and (3) an *episode* is performed. Thus, Constraint is an attribute of resource, episode or context's sub-components. Constraints can be expressed by a set containing one or more single sentences.

Figure 19 presents a scenario example. This example itself is explained later. The "Submit Order" scenario describes the interactions between the Online Broker System and its partner services, Local Supplier and International Supplier.

TITLE: ***Submit Order***
GOAL: Allow customers to find the best supplier for a given order.
CONTEXT:
PRE-CONDITION: The Broker System is online **AND** the Broker System welcome page is being displayed
ACTOR: Customer, Broker System
RESOURCES: Login page, Login information, Order
EPISODES
1. The Customer loads the login page
2. The Broker System asks for the Customer's login information
3. The Customer enters her login information
4. The Broker System checks the provided login information
5. The Broker System displays an order page
6. The Customer creates a new Order
7. **DO** the Customer adds an item to the Order **WHILE** the Customer has more items to add to the order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
10. **#** LOCAL SUPPLIER BID FOR ORDER
11. INTERNATIONAL SUPPLIER BID FOR ORDER **#**
12. PROCESS BIDS

EXCEPTIONS
1.1 **IF** Customer is not registered **THEN** REGISTER CUSTOMER
2.1 **IF** after 60 seconds **THEN** The Broker System displays a login timeout page
4.1 **IF** the Customer login information is not accurate **THEN** The Broker System displays an alert message
8.1 **IF** the order is empty **THEN** The Broker System displays an error message

Figure 19 – Example of scenario (Submit Order) in the Online Broker System.

4.1.2. Restricted-form of Natural Language

As already mentioned, scenario elements (*Title*, *Goal*, *Context*, *Resource*, *Actor*, *Episodes* and *Exception*) are written using a Restricted-form of Natural Language (RNL). In order to reduce ambiguity in natural language-based sentences, we have defined a scenario grammar for writing sentences in accordance to its conceptual model (Figure 18). Using this grammar, it is possible to write imperative (title, episode sentence or exception) and declarative (conditions or states) sentences. An imperative sentence describes actor events; and a declarative sentence describes actor or resource states.

A sentence in scenario grammar is basically defined according to the format "***Subject + Verb + Predicate***", where *subject*, *verb* and *predicate* represent the subject, main verb and objects affected by the main verb, respectively. Therefore, the sentence construction is centered on the main verb.

Table 13 shows the grammar for writing scenario elements using partial Extended-BNF (ISO/IEC 14977, 2015). The scenario model should be seen as a syntax and structural guidelines to: (1) obtain a homologous description style, (2) demonstrate the aspects that scenarios can cover and (3) facilitate the automated analysis (Leite et al., 2000).

According to the grammar described in Table 13, a *Scenario* must be described by: *Title*, *Goal*, *Context*, *Resource*, *Actor*, *Episodes* and *Exception*.

Table 13 – Scenario Grammar

TYPE	DESCRIPTION
<Scenario>	TITLE: <Title> + GOAL: <Goal> + CONTEXT: <Context> + RESOURCE: {<Resource>}$_1^N$ + ACTOR: {<Actor>}$_1^N$ + EPISODES: <Episodes> + EXCEPTION: {<Exception>}
<*Title*>	([Actor | Resource] + Action-Verb + Predicate) | Phrase
<*Goal*>	[Actor | Resource] + Verb + Predicate
<*Context*>	[GEOGRAPHICAL LOCATION: <Geographical Location>] + [TEMPORAL LOCATION: <Temporal Location>] + [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>]
<Geographical Location>	Name + [CONSTRAINT: {<Constraint>}] | <Geographical Location> <connective> <Geographical Location>
<Temporal Location>	Name + [CONSTRAINT: {<Constraint>}] | <Temporal Location> <connective> <Temporal Location>
<Pre-condition>	<expression> | <Title> | <Pre-condition> <connective> <Pre-condition>
<Post-condition>	<expression> | <Title> | <Post-condition> <connective> <Post-condition>
<expression>	((Actor | Resource) + State-Verb + Predicate) | Phrase
<connective>	AND | OR
<*Resource*>	Name + [CONSTRAINT: {<Constraint>}]
<*Actor*>	Name
<*Episodes*>	<Group> | <Episodes> <Group>
<Group>	<Sequential Group> | <Non-Sequential Group>
<Sequential Group>	<Episode><Episode> | <Sequential Group><Episode>
<Non-Sequential Group>	{<Episode>} # <Episode Series> # {<Episode>}
<Episode Series>	<Episode> <Episode> | < Episode Series><Episode>
<Episode>	<Simple Episode> | <Conditional Episode> | <Optional Episode> | <Loop Episode>
<Simple Episode>	<Id> <Episode Sentence> + [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Conditional Episode>	<Id> IF <Condition> THEN <Episode Sentence> + [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Optional Episode>	<Id> "[" <Episode Sentence> "]" + [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Loop Episode>	<Id> DO <Episode Sentence> WHILE <Condition> + [PRE-CONDITION: <Pre-condition>] + [POST-CONDITION: <Post-condition>] + [CONSTRAINT: {<Constraint>}]
<Id>	<id-chair > { (. | , | ; | :) + <id-char> } + [. | , | ; | :]
<id-char>	Letter | Digit
<Episode Sentence>	([Actor | Resource] + Action-Verb + [Direct-Object-Predicate]) | <Title>
<Condition>	<atomic sentence> | <Condition> <connective> <Condition>
<atomic sentence>	((Actor | Resource) + Linking-Verb + Predicate)| Phrase
<*Exception*>	<Id> IF <Cause> THEN <Solution> + [POST-CONDITION: <Post-condition>]
<Cause>	<atomic sentence> | <expression> | <Cause> <connective> <Cause>
<Solution>	([Actor | Resource] + Action-Verb + [Direct-Object-Predicate]) | <Title>
<Constraint>	([Actor | Resource] + [MUST] + [NOT] + Predicate) | <Title> | Phrase

In Table 13, + means composition, {x} means 0 or more occurrences of x, {x}$_1^N$ means 1 or more occurrences of x, () is used for grouping, | stands for "OR" and [x] denotes that x is optional. The following words contain only terminal symbols: *Phrase, Verb, Predicate, Name, Action-Verb, Linking-Verb, Letter,* and

Digit. The following words and phrases are terminal symbols: ***TITLE***, ***GOAL***, ***CONTEXT***, ***RESOURCE***, ***ACTOR***, ***EPISODES***, ***EXCEPTION***, ***GEOGRAPHICAL LOCATION***, ***TEMPORAL LOCATION***, ***PRE-CONDITION***, ***POST-CONDITION***, ***CONSTRAINT***, ***IF***, ***THEN***, ***WHILE***, ***DO***, ***AND***, ***OR***, ***MUST***, ***NOT***, ***"["*** and ***"]"***. Figure 19 shows how a scenario is described using the terminal and non-terminal symbols described in Table 13.

Our scenario grammar assumes that an **episode sentence** and **exception solution** are declared according to the format **"[Actor | Resource] + Action-Verb + [Direct-Object-Predicate]**", where "**Action-Verb**" express action and are the most common verbs in the present tense, and "**Direct-Object-Predicate**" refers to an object affected by the action. In the sentence "The Customer submits the Order", the work "submits" is an **Action-Verb** and the word "Order" is the **Direct-Object-Predicate**.

For example, a ***simple episode*** is described as follows:

<Id> (([Actor | Resource] + Action-Verb + [Direct-Object-Predicate]) | <Title>) +

[**PRE-CONDITION:** <Pre-condition>] +

[**POST-CONDITION:** <Post-condition>] +

[**CONSTRAINT:** {<Constraint>}]

The first element of a ***simple episode*** is the *identifier*. The second element is a ***Sentence*** that describes a situation involving users, system, environment or system's components. Optionally; the other elements are non-functional requirements (***Constraint***) related to the episode, the initial state (**Pre-condition**) before the episode is carried out, and the expected results (**Post-condition**) after the episode occurs.

An ***exception*** is described as follows:

<Id> **IF** <Cause> **THEN** (([Actor | Resource] + Action-Verb + [Direct-Object-Predicate]) | <Title>) +

[**POST-CONDITION:** <Post-condition>] +

The first element of an ***exception*** is the *identifier*. This is composed by the identifier of the *episode* followed by the number of the exception (an episode can branch several exceptions). The second element is the ***Cause*** that triggers the exception, the third element is the ***Solution*** to treat the exception, and optionally, the ***Post-condition*** attribute is the expected results after performing the *Solution*.

A **condition** may be formally defined as a logical sentence declared according to the format **"(Actor | Resource) + Linking-Verb + Predicate**". In linguistics (Cambridge, 2015), a "**Linking-Verb**" (copular verb) is a word used to link the **Subject** (Actor or Resource) of a sentence with a **Predicate** (a subject complement), such as the word "**is**" in the sentence "Feeder area is available".

Linking verbs are not followed by objects. Instead, they are followed by phrases which give extra information about the subject (e.g. noun phrases, adjective phrases, adverb phrases or prepositional phrases). Linking verbs include the conjugated form of limited number of verbs: *Be, Look, Feel, Taste, Smell, Sound, Seem, Appear, Get, Become, Grow, Stay, Keep, Turn, Prove, Go, Remain, Resemble, Run, Lie* (Usingenglish, 2015).

Like **condition**, a **State** (pre-condition and post-condition) may be formally defined as a sentence declared according to the format **"(Actor | Resource) + State-Verb + Predicate**". In linguistics (Grammaring, 2015), a "**State-Verb**" express a state which is relatively static. They include verbs of perception, cognition, the senses, emotion and state of being. In the sentence "The buffer is empty", the work "is" is a **State-Verb** and the word "empty" is the **Predicate**. State verbs are not normally used in continuous forms. Examples of state verbs include: *Appear, Be, Believe, Belong, Consider, Consist, Contain, Cost, Doubt, Exist, Fit, Hate, Hear, Have, Know, Like, Love, Matter, Mean, Need, Owe, Own, Prefer, Remember, Resemble, Seem, Suppose, Suspect, Understand, Want, Wish*.

4.1.3. Scenario Relationships-based Modularity

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. Or simply, the requirements engineers are most likely interested in a subset of scenarios. In order to face this problem, Leite et al. (2000) proposes the construction *integration scenarios* based on the existing scenarios. An ***integration scenario*** gives an overview of the relationship among several scenarios of the system, since each integration scenario episode corresponds to a **sub-scenario**. A sub-scenario details in another scenario a complex episode sentence.

Thus, the scenario language is designed with modularity in mind, mainly using a mereology operator for decomposition and the construction of ***integration***

scenarios. Modularity is considered a mechanism to deal with the scenario explosion problem (Lee et al., 1998; Leite et al., 2000).

4.1.3.1. Sequential Relationships

Besides of **integration scenario**, other *relationships* (pre-condition, post-condition, sub-scenario, exception and constraint) also provide *modularity* through the inter-connectivity among related scenarios. For example, the comprehension of an episode is facilitated by the use of natural language, well-bounded situations, and mainly through the use of ***sub-scenarios***, i.e., an episode sentence may be detailed in another scenario, or an exception may be treated by another scenario.

A scenario element is detailed in another scenario when (Leite et al., 2000):

- Common behavior is detected in several scenarios;
- A complex conditional or alternative course of action appears in a scenario; and
- The need to enhance a situation with a concrete and precise goal is detected inside a scenario.

Through these relationships it is possible to determine the order in which the scenarios should be executed. For instance, if the scenario ***X*** has among its pre-conditions the scenario ***Y***, then the last one must be executed first (precedence order).

In a scenario description, if we include the title of another scenario (UPPERCASE sentence) within the context (pre-condition or post-condition), an episode (sentence), an exception (solution) or a constraint, then, this context sub-component, episode, exception or constraint will be detailed or treated by this last scenario. Thus, the scenario language defines semantics to represent sequential relationships among scenarios. Scenarios can be connected to other scenarios through links or references, yielding a complex network of relationships:

- **Pre-condition** is a relationship defined within the ***context*** element of a scenario. If a scenario has among its pre-conditions another scenario, then the last one must be executed first.

- **Post-condition** is a relationships defined within the ***context*** element of a scenario. If a scenario has among its post-conditions another scenario, then the last one must be executed last.
- **Sub-scenario** relationship is defined when an ***episode*** (sentence) of a scenario can be described by another scenario. This allows the decomposition of complex scenarios, facilitating both its writing and understanding.
- **Exception** relationship is defined when a scenario is used to detail the treatment of an ***exception*** (solution); the scenario that treats the exception is only executed when exception's cause is triggered in the main scenario.
- **Constraint** relationship is defined when a scenario is used to detail non-functional aspects that qualify/restrict the proper execution of another, which also give us an order among the scenarios.

4.1.3.2. Non-sequential Relationships

Often in software development processes, multiple stakeholders are involved, with different needs, assumptions and points of view. But, a given group of stakeholders can be most likely interested in a specific subset of scenarios. According to Lee et al. (1998), although such subsets of scenarios might *seemingly* be independent, they are rarely truly independent in practice. Thus, scenarios also interact by complex non-sequential relationships, and in some cases these relationships are non-explicit.

So, scenarios are also related to other scenarios by explicit and non-explicit non-sequential relationships.

Explicit non-sequential relationships among scenarios are described using the structure for grouping non-sequential episodes (#<episodes series>#); i.e., if a set of episodes inside a non-sequential group are detailed in another scenarios (sub-scenario relationship), then these sub-scenarios could be executed in an indistinct order or concurrently. In Figure 22, the episodes 10 and 11 of the main execution flow reference sub-scenarios described in Figure 23. These sub-scenarios are explicitly described to be executed in an indistinct order or concurrently.

In some cases, the given scenarios could interact by **non-explicit and non-sequential relationships**; often, they communicate by concurrency, which can lead to erroneous situations such as deadlocks. From the concurrency perspective, a set of scenarios can be considered as a set of ***concurrently*** executing threads, and they could interact or compete with each other by ***communication channels*** or ***shared resources***.

In practice, it is very difficult to identify non-sequential relationships among scenarios, because most of the proposed languages to write scenarios do not provide:

- Constructs or semantics to represent explicitly the relationships among scenarios;
- Heuristics to find non-explicit relationships based on concurrency characteristics (e.g. non-determinism and synchronization by shared resources);
- Heuristics to assist the developer in making explicit non-sequential relationships.

4.1.3.3. Heuristics to Find Non-explicit and Non-sequential Relationships

An heuristic for finding non-explicit relationships is shown in this sub section. It uses information of scenario descriptions and the scenario model for making explicit non-sequential relationships among scenarios.

This heuristic could assist the developers in identifying concurrency opportunities since initial requirements engineering activities, and requirement engineers in detecting defects arose from interactions among related scenarios.

In a concurrent system, local processes are first developed, and it has particular characteristics such as ***non-deterministic*** execution and ***synchronization*** between processes. These characteristics arise from the possibility of communication between process, which can be via ***communication channels*** or ***shared resources***, resulting in complex interactions.

In the scenario language, two or more scenarios are likely related when they share common portions in their descriptions, i.e., they involve the participation of common actors, they access shared resources or they are executed in the same context. Leite et al. (2005) used the concept of **Proximity Index** to more detailed

comparisons between any two scenarios with obscure and poorly defined borders. It is defined by:

Let $\mathbf{I_{ij} = (\alpha * C\cap_{ij} + \beta * A\cap_{ij} + \Upsilon * R\cap_{ij}) / (\alpha * C\cup_{ij} + \beta * A\cup_{ij} + \Upsilon * R\cup_{ij})}$ be the proximity index of Scenarios $\mathbf{S_i}$ and $\mathbf{S_j}$; where:

$\mathbf{\alpha, \beta, \Upsilon}$ are weight factors.

$\mathbf{C\cap_{ij}}$ = | Context (S_i) ∩ Context (S_j) |;

$\mathbf{A\cap_{ij}}$ = | Actor (S_i) ∩ Actor (S_j) |;

$\mathbf{R\cap_{ij}}$ = | Resource (S_i) ∩ Resource (S_j) |;

$\mathbf{C\cup_{ij}}$ = | Context (S_i) ∪ Context (S_j) |;

$\mathbf{A\cup_{ij}}$ = | Actor (S_i) ∪ Actor (S_j) |;

$\mathbf{R\cup_{ij}}$ = | Resource (S_i) ∪ Resource (S_j) |;

Actor ($\boldsymbol{S_k}$): Actors of scenario **k**;

Resource ($\boldsymbol{S_k}$): Resources of scenario **k**;

Context ($\boldsymbol{S_k}$): Context of scenario **k**.

As the **first step** of the heuristic for finding non-explicit non-sequential relationships among scenarios (see Heuristic 1), we filter sequentially and explicit non-sequentially related scenarios.

As the **second step** of the Heuristic 1, we adapted the **Proximity Index** among any two scenarios defined in (Leite et al., 2005), by considering only common actors or shared resources.

In this case, actors and resources have the same importance because two scenarios might interact by common actors or shared resources ($\beta = \Upsilon = 1$). Thus, if two scenarios have common actors or share resources, then, they could be related to each other. If the **Proximity Index** is higher or equal than 0.5, then there is an indication that scenarios need to be compared in a more detailed way.

Let $\mathbf{I_{ij} = MAX ((A\cap_{ij} / A\cup_{ij}), (R\cap_{ij} / R\cup_{ij}))}$ be the proximity index of Scenarios $\mathbf{S_i}$ and $\mathbf{S_j}$; where:

MAX (x, y): Find *maximum* of *x* and *y*.

As the **third step** of this heuristic, each pair of two scenarios with higher proximity index is compared in more detail. This comparison is needed to determine whether they interact by **non-determinism** or **synchronization** constraints; that is:

- **Non-determinism**: It compares **pre-conditions** to determine whether there is a **non-deterministic** execution or not. For example, when a pre-condition described in a scenario $\mathbf{S_i}$ appears like pre-condition in another scenario $\mathbf{S_j}$, then, $\mathbf{S_i}$ and $\mathbf{S_j}$ might interact concurrently.
- **Synchronization**: It compares **pre-conditions** against **post-conditions** to determine whether there is **synchronization** or not. For example, when a pre-condition described in a scenario $\mathbf{S_i}$ appears like post-condition in another scenario $\mathbf{S_j}$, and a pre-condition described in $\mathbf{S_j}$ appears like post-condition in $\mathbf{S_i}$, then, $\mathbf{S_i}$ and $\mathbf{S_j}$ might interact concurrently.

In ***Heuristic 1*** (Figure 20), we list some general criteria to make explicit potentially concurrent scenarios ($\mathbf{S_i}$ and $\mathbf{S_j}$), since **non-deterministic** and **synchronization** perspectives. In order to compare two scenario elements (e.g. two goals), or verify the similarity between an item and the items of a set (e.g. intersection between two set of pre-conditions), we use Levenshtein's distance (Levenshtein, 1966).

In Heuristic 1:

- **Seq-Related**(S_i, S_j): scenario **i** and scenario **j** are sequentially related by: pre-condition, post-condition, constraint, sub-scenario or exception.
- **Explicit-Non-Seq-Related**(S_i, S_j): scenario **i** and scenario **j** are non-sequentially related by non-sequential group construct: #<episode series>#.
- **Pre-Cond** (S_k): {pre-conditions in the context of scenario **k** ∪ pre-conditions in the episodes of scenario **k**};
- **Post-Cond** (S_k): {post-conditions in the context of scenario **k** ∪ post-conditions in the episodes of scenario **k**};

Heuristic 1: Making Explicit Non-sequential Relationships
Input: Scenario S_i and Scenario S_j
Output: are S_i and S_j potentially concurrent? : {YES or NOT}
Begin:
1. **IF Seq-Related**(S_i, S_j) **THEN** ***Return*** **NOT**;
2. **IF Explicit-Non-Seq-Related**(S_i, S_j) **THEN** ***Return*** **NOT**;
3. Calculate the proximity index for S_i and S_j: I_{ij} = MAX $((A\cap_{ij} / A\cup_{ij}), (R\cap_{ij} / R\cup_{ij}))$;
4. **IF** $I_{ij} \geq 0.5$ **THEN** determine whether S_i and S_j are concurrent by *non-determinism*:
   →**IF** | Pre-Cond (S_i) ∪ Pre-Cond (S_j) | = | Pre-Cond (S_i) ∩ Pre-Cond (S_j) | **THEN**
   S_i and S_j are simultaneously enabled by the same pre-condition;
   S_i and S_j are potentially concurrent;
   Return **YES**;
5. **IF** $I_{ij} \geq 0.5$ **THEN** determine whether S_i and S_j are concurrent by *synchronization*:
   →**IF** | Pre-Cond (S_i) ∩ Post-Cond (S_j) | ≥ **1 AND** | Post-Cond (S_i) ∩ Pre-Cond (S_j) | ≥ **1 THEN**
   S_i and S_j are simultaneously executed;
   S_i and S_j are potentially concurrent;
   Return YES;
6. ***Return*** **NOT**;
End

Figure 20 - Making Explicit Non-sequential Relationships (Heuristic 1).

In order to improve the reliability of systems initially specified as scenarios representations, the identification of non-explicit relationships among scenarios makes it possible to perform rigorous analysis focusing on related scenarios and achieve a more consistent and more correct requirements specification. This is especially important for systems involving concurrent, asynchronous, distributed, non-deterministic or parallel processes, such as distributed web services, multi-agent systems, manufacturing systems or shared memory-based systems.

4.1.4. Running Example

This section describes a set of scenarios for describing a system that involves sequential and non-sequential relationships.

In the ***Online Broker System***, the ***Broker System*** interacts with its partner services: ***Local Supplier*** and ***International Supplier***. The system under consideration is an *Online Broker System*. The goal of the system is to allow customers to find the best supplier for a given order. A customer fills up an online order form and after submission; the system broadcasts it to local and international suppliers. Each supplier after examining the order may decide to decline or submit a bid. A local supplier needs to add taxes to the order total, while an international supplier needs to ensure an order does not include items restricted for export. Submitted bids are sent back to the broker to be shown to the customer, **who eventually asks the system to proceed with a bid**. The full

scenarios of the "Online Broker System" example are shown in (Somé, 2010) using a use case language based on Cockburn's template (Cockburn, 2001).

In order to understand the execution order of a set of related scenarios, it is necessary to identify the ***main scenario*** of this set (or main scenarios). A ***main scenario*** will be the scenario that does not require any other scenario of the set, or that reference in its description to other scenarios. According to Almentero et al. (2014), we first determine the relationship between the scenarios of the set, and from identified relationships we will establish an execution order between them.

In the "Online Broker System", the ***Submit Order*** scenario is a ***main scenario*** because it precedes all others, and it is related to other scenarios by sequential and explicit non-sequential relationships. In the original version shown in (Somé, 2010), it is not obvious to perceive the relationships among related scenarios and that ***Process Bids*** scenario is a scenario executed after querying the customer (See Figure 21). The ***Process Bids*** scenario (Figure 23) is referenced inside ***Supplier*** scenarios (Somé, 2010). Other relationship that is difficult to perceive is the sequential relationship between "***Register Customer***" (Figure 23) and the *main scenario* "***Submit Order***". The meaning of the relationship is that scenario "***Register Customer***" extends scenario "***Submit order***" when condition "*Customer is not registered*" holds.

TITLE: ***Submit Order***
SYSTEM UNDER DESIGN: Broker System
PRE-CONDITION: The Broker System is online **AND** the Broker System welcome page is being displayed
SUCCESS POST-CONDITION: An Order has been broadcasted
STEPS
1. The Customer loads the login page
2. The Broker System asks for the Customer's login information
3. The Customer enters her login information
4. The Broker System checks the provided login information
5. The Broker System displays an order page
6. The Customer creates a new Order
7. **Repeat while** The Customer has more items to add to the order
   7.1 The Customer adds an item to the Order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
10. Enable in parallel use cases Local Supplier bid for order, International bid for order

ALTERNATIVES
2a. after 60 seconds
   2a1. The Broker System displays a login timeout page
4a. The Customer login information is not accurate
   4a1. **GOTO** Step 2.
8a. The Order is empty
   8a1. The Broker System displays an error page

EXTENSION POINTS
STEP 1. login page loaded

Figure 21 - "Submit Order" use case in the Online Broker System (Somé, 2010).

Using the scenario language proposed in this work, we re-described the ***Submit Order*** scenario to make explicit the sequential relationships by: (1) adding

a last episode, which references the ***Process Bids*** scenario through *sub-scenario relationship* (See Figure 22); and (2) mapping the extension point into an exception with references the ***Register Customer*** scenario through *exception relationship*.

TITLE: ***Submit Order***
GOAL: Allow customers to find the best supplier for a given order.
CONTEXT:
PRE-CONDITION: The Broker System is online **AND** the Broker System welcome page is being displayed
ACTOR: Customer, Broker System
RESOURCES: Login page, Login information, Order
EPISODES
1. The Customer loads the login page
2. The Broker System asks for the Customer's login information
3. The Customer enters her login information
4. The Broker System checks the provided login information
5. The Broker System displays an order page
6. The Customer creates a new Order
7. **DO** the Customer adds an item to the Order **WHILE** the Customer has more items to add to the order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
10. **#** LOCAL SUPPLIER BID FOR ORDER
11. INTERNATIONAL SUPPLIER BID FOR ORDER **#**
12. PROCESS BIDS

EXCEPTIONS
1.1 **IF** Customer is not registered **THEN** REGISTER CUSTOMER
2.1 **IF** after 60 seconds **THEN** The Broker System displays a login timeout page
4.1 **IF** the Customer login information is not accurate **THEN** The Broker System displays an alert message
8.1 **IF** the order is empty **THEN** The Broker System displays an error message

Figure 22 - Description of scenario "Submit Order" in the Online Broker System.

In Figure 22, the episodes *10* (LOCAL SUPPLIER BID FOR ORDER), *11* (INTERNATIONAL SUPPLIER BID FOR ORDER), *12* (PROCESS BIDS), and exception *1.1* (REGISTER CUSTOMER) are detailed in another scenarios. Figure 22 shows the sequential interaction among scenarios by sub-scenario (PROCESS BIDS) and exception (REGISTER CUSTOMER) relationships, and non-sequential relationships by explicit concurrency construct (SUPPLIERS).

PROCESS BIDS, REGISTER CUSTOMER, LOCAL SUPPLIER BID FOR ORDER and INTERNATIONAL SUPPLIER BID FOR ORDER are presented in Figure 23 and detailed in Appendix 1. PROCESS BIDS references sequentially to HANDLE PAYMENT scenario.

TITLE: ***Local Supplier bid for order***
GOAL: Submit a bid
CONTEXT: Create a Bid for an Order
PRE-CONDITION: An Order has been broadcasted
POST-CONDITION: Local Supplier has bidden
ACTOR: Local Supplier, Broker System
RESOURCES: Order, Bid
EPISODES
1. Local Supplier receives the Order and examines it
2. Local Supplier determines the applicable taxes to the order and creates a bid
3. Local Supplier submits a Bid for the Order
4. The Broker System sends the Bid to the Customer
EXCEPTIONS
1.1 **IF** Local Supplier can not satisfy the Order **THEN** Local Supplier passes on the Order

TITLE: ***International Supplier bid for order***
GOAL: Submit a bid
CONTEXT: Create a Bid for an Order
PRE-CONDITION: An Order has been broadcasted
POST-CONDITION: International Supplier has bidden
ACTOR: International Supplier, Broker System
RESOURCES: Order, Bid
EPISODES
1. International Supplier receives the Order and examines it
2. International Supplier submits a Bid for the Order
3. The Broker System sends the Bid to the Customer
EXCEPTIONS
1.1 **IF** The Order includes items restricted for exportation **THEN** International Supplier passes on the Order
1.2 **IF** International Supplier can not satisfy the Order **THEN** International Supplier passes on the Order

TITLE: ***Process Bids***
GOAL: Process a bid
CONTEXT: Process a Bid for an Order
PRE-CONDITION: Local Supplier has bidden **OR** International Supplier has bidden
ACTOR: Customer, Broker System
RESOURCES: Order, Bid
EPISODES
1. Customer examines the bid
2. Customer signals the system to proceed with bid
3. HANDLE PAYMENT
4. System put an order with the selected bidder

TITLE: ***Register Customer***
GOAL: Register Customer
CONTEXT: login page loaded
PRE-CONDITION:
POST-CONDITION:
ACTOR: Customer, Broker System
RESOURCES: registration operation, name, date of birth, address, login information
EPISODES
1. Customer selects registration operation
2. Broker System asks for Customer name, date of birth and address
3. Customer enters registration information
4. Broker System validates Customer information
5. Broker System generate login information for Customer
EXCEPTIONS
4.1. IF Customer registration information is not valid THEN Broker System displays registration failure page

TITLE: ***Handle Payment***
GOAL: Handle Payment
CONTEXT: Handle payment for a Bid
PRE-CONDITION:
POST-CONDITION:
ACTOR: Customer, Broker System, Payment System
RESOURCES: Credit card information
EPISODES
1. The Broker System asks the Customer for Credit Card information
2. The Customer provides her Credit Card information
3. The Broker System asks a Payment System to process the Customer's Payment
4. The Broker System displays an acknowledgement message to the Customer
EXCEPTIONS
1.1 **IF** The Customer Payment is denied **THEN** The Broker System displays a payment denied page

Figure 23 – Scenarios of the "Online Broker System".

In this example, from the *main scenario* (Submit Order), it is possible to identify the sequentially (PROCESS BIDS, REGISTER CUSTOMER) and explicit non-sequentially related scenarios (LOCAL SUPPLIER AND INTERNATIONAL SUPPLIER).

In most of projects, it is difficult to identify the non-explicit relationships among scenarios, mainly, non-sequential relationships among them.

For example, if we do not have any scenario referencing explicitly other scenarios, it will be difficult to perceive that ***Suppliers'*** scenarios are non-sequentially related to each one. So, in order to identify non-explicit relationships of "Online Broker System" scenarios, we apply the heuristic for finding non-explicit relationships (described in Heuristic 1) to explore any two potentially related scenarios.

As the first step of the heuristic, we calculate the proximity index among any two scenarios. For example, we chose "Local Supplier for Bid" **(S_1) and**

"International Supplier for Bid" **(S_2)** to be explored, and they have a degree of proximity high (proximity index = 1). Therefore, they must be analyzed more deeply.

As the second step, we detect that they are enabled by the same pre-condition, and then they are non-sequentially related by non-determinism feature.

Table 14 shows the results of proximity index (equal to 1) and scenarios S_1 and S_2 are related by non-explicit non-sequential relationships (Non-determinism).

Table 14 – Proximity Index between Scenarios of the Online Broker System

S_i	S_j	$A\cap_{ij}$	$A\cup_{ij}$	$R\cap_{ij}$	$R\cup_{ij}$	I_{ij}	Non-determinism			Synchronization
							Goal	*Temp_Loc*	*Pre-Condition*	*Pre-Condition & Post-Condition*
S_1	S_2	2	4	2	2	1	Similar	---	YES	---

In Figure 23, the "Local Supplier for Bid" and "International Supplier for Bid" specify as common pre-condition the availability of the *"An order has been broadcasted"*. Thus, these scenarios interact by ***shared resources***.

We identified the non-explicit relationships, because they can be used to perform early concurrent (potentially concurrent) system analysis to detect potential defects due to concurrency at early software development activities.

4.2. Pre-processing Scenarios

In order to improve the efficacy of scenario transformation algorithm and the accuracy of NLP analysis tools, it is necessary to remove the irrelevant information and formatting symbols, such as URLs, HTML tags, parenthesized comments and bullets. According to Liu et al. (2014), the noise from the input document may affect the parsing accuracy. This is a general process applicable to any document.

Therefore, the steps to clean scenarios of these possible noise elements are described below:

- **Removal of Empty Line**: There is no empty line in the scenario.
- **Removal of Capitalization:** often it is convenient to lower case every character.
- **Removal of Brackets:** Text between brackets within a sentence is replaced by empty character. There are various bracket symbols: Parentheses "()", Square Brackets "[]" and Curly Braces "{}".

- **Removal of URLs**: URLs and hyperlinks within a sentence like comments or reviews should be removed.
- **Removal of HTML Markup:** HTML tags within a sentence should be removed.
- **Removal Punctuation:** For NLP analysis, all the punctuation marks and bullets according to the priorities should be dealt with. For example: "!", "#", "?", "•" are important punctuations that need to be removed and replaced by a white space character.
- **Apostrophe Lookup:** According to Bansal (2014), to avoid any word sense disambiguation in text, it is recommended to maintain proper structure in it and to abide by the rules of context free grammar. When apostrophes are used, chances of disambiguation increase. For example "it's is a contraction for it is or it has".

We utilized regular expression matching to perform the filtering tasks. The last step (**Apostrophe Lookup**) was not considered in our pre-processing process because it is fairly domain dependent and a challenging topic in NLP research.

4.3. Deriving Petri-Nets

After constructing scenarios, it is possible to automatically derive Petri-Net formal specifications. In our approach, each scenario sentence (imperative or declarative) is translated into a Petri-Net node (transition or place, respectively). These Petri-Net nodes are linked by arcs giving rise to a Petri-Net model. Each translated scenario defines components of the initial system design.

4.3.1. Transforming Scenarios into Petri-Nets

We assume that a ***scenario*** **S**: (1) *starts* at an *idle state* with all necessary *resources*, *pre-conditions* or *constraints*; (2) *performs* a collection of partially ordered *event* occurrences (episodes or exceptions), each guarded by a set of *conditions* (pre-conditions, post-conditions, or causes) and restricted by a set of *constraints*; and (3) *returns* to the *idle state* releasing the *resources*, *pre-conditions* (if it is not returned by some previous event) or *constraints* after completion (adapted from Cheung et al. , 2006).

A Petri-Net **PN** is derived from a scenario **S** as follows: We identify the *event occurrences* (episodes and exceptions) and their *pre-conditions* (or causes), *constraints* and *post-conditions*. For each *event*, a *transition* is created for denoting the location of *event occurrence*. *Input places* are created to denote the locations of its *pre-conditions, causes* and *constraints* (They restrict but do not impede – *TRUE*). *Output places* are created to denote the location of its *post-conditions*. Event labels, condition labels and constraint labels are assigned to these transitions and places accordingly. The initial marking $\mathbf{M_0}$ of the **PN** is then created to denote the initial state, in which tokens are added into input places that represent *pre-conditions, causes* or *constraints*. Execution of the scenario begins at this initial marking which semantically means the system initial state, including the availability of all resources, pre-condition, causes or constraints. It ends at the same marking that semantically means the release of these resources, pre-conditions, causes or constraints.

As the ***first step*** of the method for *Transforming a Scenario into an Equivalent Petri-Net* (***Method 1*** in Figure 26), we define mapping rules to translate scenario elements (Title, Goal, Context, Resource, Actor, Episodes, Exception) into Petri-Net elements (*transition*, *place* and *arc*).

For each scenario element, a sub Petri-Net which contains places, transitions and arcs is derived. The different mapping rules to derive a sub Petri-Net from a scenario element are described using a structure composed of left and right hand sides (LHS and RHS). LHS is the conditional part of the rule (scenario element), and RHS is the expected result of the rule (sub Petri-Net).

Table 15, Table 16, Table 17, Table 18 and Table 19 define the mapping rules for initial state, episodes, exception, concurrency constructs and final state of a scenario, respectively. In LHS side (Scenario), "**e**" is an episode and "**ex**" is an exception. In RHS side (Sub Petri-Net), "**t**" is a transition (with the name attribute), "**p**" is a place (with the name and number of tokens attributes) and "**a**" is an arc (with source and target attributes). Below, we detail these mapping rules.

In order to preserve the event sequences described within the main flow (episodes) and exceptional flows of a scenario, we add appropriate *Input dummy place* and *Output dummy place* to the sub Petri-Nets derived from scenario elements. These dummy places are used for linking sub Petri-Nets derived from sequential events (e.g. episode 1 and episode 2 of the main flow of episodes).

A *Dummy transition* is added to the sub Petri-Nets derived from scenario initial state and final state. It represents an initial event or a final event derived from the main flow of episodes, i.e., a scenario *initial state* or *final state* is mapped into a sub Petri-Net composed of a *dummy transition* and its corresponding input and output dummy places.

Table 15 – Transforming Scenario Triggering

Rule	Transform Scenario Triggering – Initial State
	When
LHS (Scenario)	Title, Resources, Context = {Constraint, Pre-condition }
	Then
RHS (sub Petri-Net)	1. Generate: →Dummy Transition t with: t.name = "DUMMY"; →Input dummy Place p of t with p.name = "START", representing the Title and Resources; →Output dummy Place p of t; →Link Input and Output dummy Place to Dummy transition t; 2. For every Constraint c in {Context ∪ Resources}, generate: →Input Place p of t with: p.name = c.name; p.tokens = 1; →Output Arc a of t with: a.source = t; a.target = p; 3. For every Pre-condition pre in Context, generate: →Input Place p of t with: p.name = pre.name; p.tokens = 1; 4. Returns sub Petri-Net;
	End

Table 16 – Transforming Episode

Rule	**Transform Episode**
	When
LHS (Scenario)	Episode **e** = {Id, Sentence, Type, Condition, Constraint, Pre-condition, Post-condition}
	Then
RHS (sub Petri-Net)	1. Generate: →Transition, **t** with t.name = **e**.Sentence; →Input dummy Place of **t**; →Output dummy Place of **t**; →Link Input and Output dummy Place to Dummy transition t; 1.1. **IF e**.Type = "*CONDITIONAL*" OR *"OPTIONAL"*, generate: →Dummy Transition **t_else** with: **t_else**.name = "ELSE"; →Input Arc **a** of **t_else** with: **a**.source = Input dummy Place of **t**; →Output Arc **a** of **t_else** with: **a**.target = Output dummy Place of **t**; 1.2. **IF e**.Type = "*LOOP*", generate: →Dummy Transition **t_iteration** with: **t_ iteration**.name = "ELSE"; →Input Arc **a** of **t_iteration** with: **a**.source = Output dummy Place of **t**; →Output Arc **a** of **t_iteration** with: **a**.target = Input dummy Place of **t**; 2. For every Condition **c** in **e**, generate: →Input Place **p** of **t** with: **p**.name = **c**.name; **p**.tokens = 1; →Output Arc **a** of **t** with: **a**.source = **t**; **a**.target = **p**; 3. For every Constraint **c** in **e**, generate: →Input Place **p** of **t** with: **p**.name = **c**.name; **p**.tokens = 1; →Output Arc **a** of **t** with: **a**.source = **t**; **a**.target = **p**; 4. For every Pre-condition **pre** in **e**, generate: →Input Place **p** of **t** with: **p**.name = **pre**.name; **p**.tokens = 1; 5. For every Post-condition **post** in **e**, generate: →Output Place **p** of **t** with: **p**.name = **post**.name; 6. Returns sub **Petri-Net**;
	End

Table 17 – Transforming Concurrency Construct

Rule	**Transform Concurrency Construct**
	When
LHS (Scenario)	Episode **e1** and Episode **e2**
	Then
RHS (sub Petri-Net)	1. IF **e1**.sentence starts with "#", generate: →Dummy Transition **t** with: t.name = "FORK"; →Input dummy Place **p** of **t**; →Output dummy Place **p** of **t**; →Link Input and Output dummy Place to Dummy transition t; 2. IF **e2**.sentence ends with "#", generate: →Dummy Transition **t** with: **t**.name = "JOIN"; →Input dummy Place **p** of **t**; →Output dummy Place **p** of **t**; →Link Input and Output dummy Place to Dummy transition t; 3. Returns sub **Petri-Net1** for **e1** and **Petri-Net2** for **e2**;
	End

Table 18 – Transforming Exception

Rule	**Transform Exception**
	When
LHS (Scenario)	Episode **ex** = {Id, Cause, Solution, Post-condition}
	Then
RHS (sub Petri-Nets)	1. Generate: →Transition **t** with: t.name = **ex**.solution; →Input dummy Place **p** of **t**; →Output dummy Place **p** of **t**; →Link Input and Output dummy Place to Dummy transition t; 2. For every Cause **c** in **ex**, generate: →Input Place **p** of **t** with: **p**.name = **c**.name; **p**.tokens = 1; 3. For every Post-condition **post** in **ex**, generate: →Output Place **p** of **t** with: **p**.name = **post**.name; 4. Returns sub **Petri-Net**;
	End

Table 19 – Transforming Scenario Completion

Rule	**Transform Scenario Completion – Final State**
	When
LHS (Scenario)	Context = {Post-condition }
	Then
RHS (sub Petri-Net)	1. Generate: →Dummy Transition **t** with: t.name = "DUMMY"; →Input dummy Place **p** of **t**; →Output dummy Place **p** of **t** with p.name = "FINISH"; →Link Input and Output dummy Place to Dummy transition t; 2. For every Post-condition **post** in **Context**, generate: →Output Place **p** of **t** with: **p**.name = **post**.name; **p**.tokens = 1; 3. Returns sub **Petri-Net**;
	End

Figure 24 depicts the visual transformation (LHS→RHS) of a simple episode into Petri-Net elements. In this example, a simple episode (Submit Order

Scenario) is mapped into a transition with an input dummy place and an output dummy place of the transition.

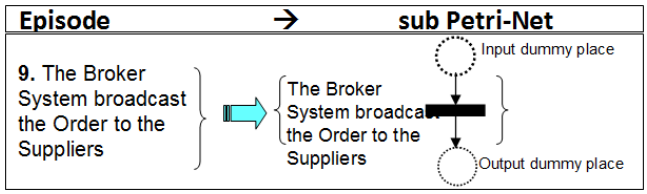


Figure 24 – Transforming Simple Episode

The visual transformation (LHS→RHS) from Scenario elements into Petri-Net elements is depicted in

Figure 25. These transformations perform the tasks (mapping rules) defined in Table 15, Table 16, Table 17, Table 18 and Table 19.

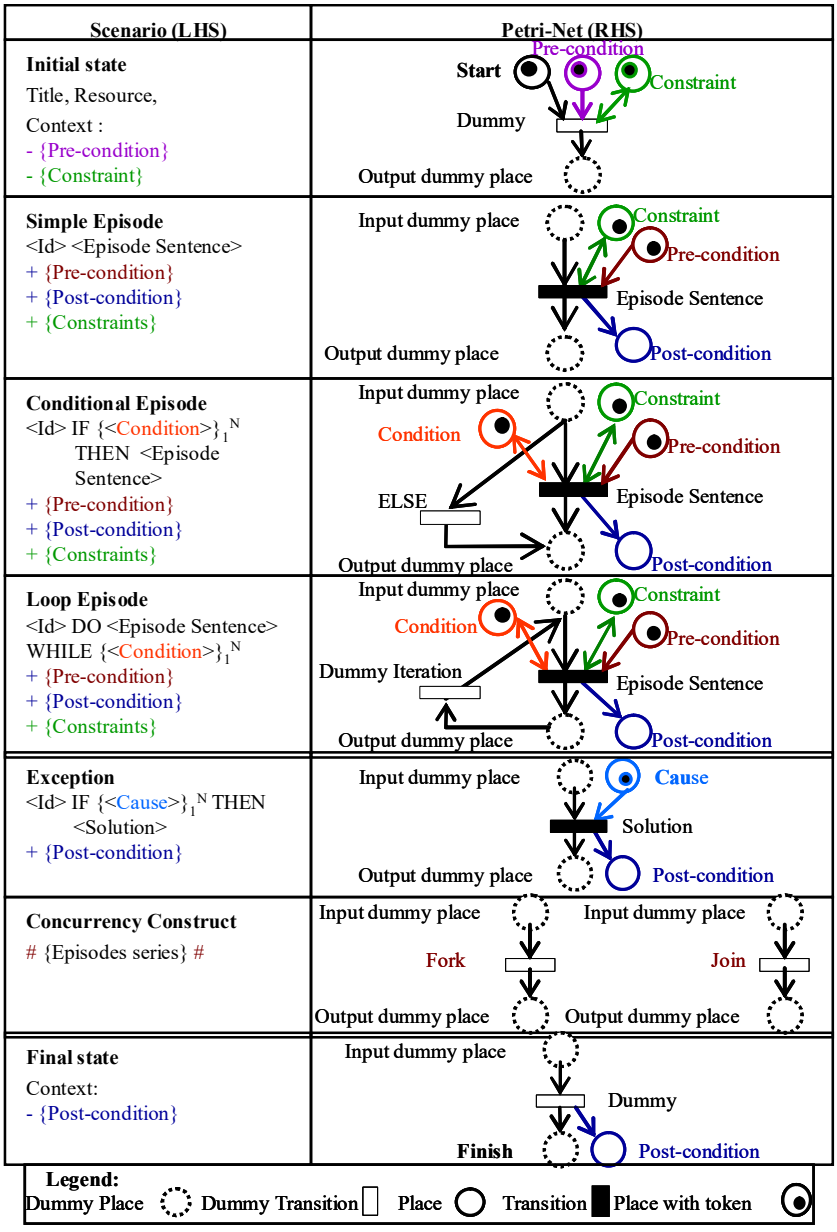


Figure 25 - Mapping scenario constructs into Petri-Net elements.

As the ***second step*** of the ***Method 1*** (Figure 26), the sub Petri-Nets generated from scenario elements are composed into a whole Petri-Net by *Fusion Place* or *Modified Fusion Place* operations.

Definition 6.1 (Fusion Place): A sub Petri-Net can be fused with other sub Petri-Net by fusing the *output dummy place* of the first sub Petri-Net into the *input dummy place* of the last sub Petri-Net.

Definition 6.2 (Modified Fusion Place): Any sub Petri-Net can be fused with other sub Petri-Net by fusing at least a *common place* among them. For example, two sub Petri-Nets derived from different episodes can be fused, if they have a common place that represents a common pre-condition.

Method 1: Transform Scenario into Petri-Net
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Output: Petri-Net ***PN*** **=** (P, T, F, W, M_0)**;**
Begin:
1. Clean Scenario **S** from unnecessary information (Pre-processing)
2. Generate a sub Petri-Net for ***scenario triggering***: Apply Transforming *Initial State* rule (Table 15);
3. **For** every ***episode*** generate a sub Petri-Net:
→**IF** episode sentence starts with "**#**" **THEN** Apply Transforming Concurrency Construct rule (Table 17);
→**IF** episode is Simple **THEN** Apply Transforming Episode rule (Table 16);
→**IF** episode is Conditional **THEN** Apply Transforming Episode rule (Table 16);
→**IF** episode is Optional **THEN** Apply Transforming Episode rule (Table 16);
→**IF** episode is Loop **THEN** Apply Transforming Episode rule (Table 16);
→**IF** episode sentence ends with "**#**" **THEN** Apply Transforming Concurrency Construct rule (Table 17);
4. For every ***exception*** generate a sub Petri-Net:
→Apply Transforming Exception rule (Table 18);
5. Generate a sub Petri-Net for ***scenario completion***: Apply Transforming *Final State* rule (Table 19);
6. Link sub Petri-Nets of *exceptions* to sub Petri-Nets of *branching episodes*;
→Apply Fusion Place to sub Petri Nets from episode and exception (Definition 6.1);
7. **Link** sub Petri-Nets between a *fork and a join transitions* as concurrent sub Petri-Nets;
→Apply *Fusion Place* to sub Petri-Nets from ***fork*** and episode (Definition 6.1);
→Apply *Fusion Place* to sub Petri-Nets from episode and ***join*** (Definition 6.1);
8. **Compose** the sub Petri-Nets into a *complete* ***Petri-Net***:
→**For** every sub Petri-Net
→Apply *Fusion Place* operation, following the precedence order (Definition 6.1);
→Apply *Modified Fusion Place* operation (Definition 6.2);
9. FOR every *input place* of the *first transition* (initial state):
→**IF** *input place* has not *input arcs* **THEN Link** *last transition* to the *input place*;
10. Return **Petri-Net**;
End

Figure 26 – Transform Scenario into Petri-Net (Method 1).

4.3.2. Integrating Petri-Nets

For every scenario and its related scenarios, we generate partial Petri-Nets in order to integrate these partial Petri-Nets into a consistent whole ***Integrated Petri-Net***. The ***Integrated Petri-Net*** reflects exactly the original properties of the synthesized Petri-Nets (Demonstrated in Section 4.3.4).

In the proposed scenario language, scenarios are related to other scenarios by explicit sequential relationships (pre-condition, post-condition, constraint, sub-scenario or exception). When a *scenario* is chosen to be a ***main scenario***, and translated into a ***main Petri-Net***, the referenced scenarios (sequentially) are mapped into *input places* (pre-conditions or constraints), *output places* (post-conditions) or *transitions* (episodes' sentence or exceptions' solution).

In this case, a main scenario is the starting point to find the related scenarios.

As the ***first step*** of the method for *integrating Petri-Nets (**Method 2*** in Figure 27*)*, each sequentially related scenario is translated into a Petri-Net. After it, each one of these Petri-Nets must be replaced into the corresponding place or transition of the *main Petri-Net*. Our first step is the *substitution of places or transitions*.

Definition 6.3 (Substitution Transition): Any *transition* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the *input dummy place* of the *transition* is fused with the first *input dummy place (Start)* of the *replacing Petri-Net* and the *output dummy place* of the *transition* is fused with the last *output dummy place (Finish)* of the *replacing Petri-Net.*

Definition 6.4 (Substitution Input Place): Any *input place* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the last *output dummy place (Start)* of the *replacing Petri-Net* is fused with the *input place*.

Definition 6.5 (Substitution Output Place): Any *output place* (not dummy) of a Petri-Net can be replaced by any other Petri-Net. Then the first *input dummy place (Start)* of the *replacing Petri-Net* is fused with the *output place*.

Scenarios are also related to other scenarios by explicit and non-explicit non-sequential relationships (Indistinct order or Concurrency).

Explicit non-sequential relationships among scenarios are described using the structure for grouping non-sequential episodes (#<episodes series>#). If a ***main scenario*** is mapped into a ***main Petri-Net***, the explicit non-sequentially related scenarios (episodes between a Concurrency Construct) are mapped into *transitions*.

Non-explicit and non-sequential relationships among scenarios are found by analyzing *common actors* or *shared resources* (See Section 4.1). If a ***main scenario*** is mapped into a ***main Petri-Net***, the interaction with non-explicit and

non-sequentially related scenarios is described by common *pre-conditions* or *post-conditions*, these common conditions are mapped into *input places* or *output places*.

As the ***second step*** of the method for *integrating Petri-Nets (**Method 2*** in Figure 27*)*, each non-sequentially related scenario is translated into a Petri-Net. Among the Petri-Nets, there are *common places* (with the same labels) that denote the same *pre-condition* or *post-condition*, and they need to be uniquely represented from the system point of view (Cheung et al., 2006). Our second step is basically the *substitution of transitions* (episodes referencing explicit non-sequential scenarios) and *fusion of common places* (non-explicit non-sequential scenarios interact by common conditions).

Definition 6.6 (Concurrent Fusion Place): Any Petri-Net can be fused with other Petri-Net by fusing at least a *common place* (from *pre-condition* or *post-condition*) among them.

Method 2: Integrate Petri-Nets
Input: Main Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Output: Integrated Petri-Net ***IPN*** **=** (P, T, F, W, M_0)
Begin:
1. Derive ***Main Petri-Net*** from the Main ***Scenario*** (*Method 1*);
2. Identify *sequential relationships* from the *Main* ***Scenario*** by *Pre-condition. Post-condition, Constraint, Sub-scenario or Exception*;
3. Identify explicit *non-sequential relationships* from the *Main* ***Scenario*** by analyzing Concurrency Constructs;
4. Identify non-explicit *non-sequential relationships* from the *Main* ***Scenario*** by common *Pre-condition* or *Post-condition*;
5. Obtain a whole *Integrated Petri-Net* from the *Main* ***Petri-Net***:
→**For** every *scenario* in *sequentially related scenarios*:
→Transform *scenario* into a *Petri-Net* (*Method 1*);
→**IF** *current* Petri-Net represents a *Sub-scenario* or *Exception* in *Main* Scenario **THEN**:
→*Substitute* the corresponding "*Transition*" of the ***Main Petri-Net*** (Definition 6.3);
→**IF** *current* Petri-Net represents a *Pre-Condition* or *Constraint* in *Main* Scenario **THEN**:
→*Substitute* the corresponding "*Input Place*" of the ***Main Petri-Net*** (Definition 6.4);
→**IF** *current* Petri-Net represents a *Post-Condition* in Main Scenario **THEN**:
→*Substitute* the corresponding "*Output Place*" of the ***Main Petri-Net*** (Definition 6.5);
→**For** every *scenario* in explicit non-*sequentially related scenarios*:
→Transform *scenario* into a *Petri-Net* (*Method 1*);
→*Substitute* the corresponding "*Transition*" of the ***Main Petri-Net*** (Definition 6.3);
→**For** every *scenario* in non-explicit non-*sequentially related scenarios*:
→Transform *scenario* into a *Petri-Net* (*Method 1*);
→*Fuse the common places* between the *current Petri-Net* and ***Main Petri-Net*** (Definition 6.6);
6. Return integrated ***Main Petri-Net***;
End

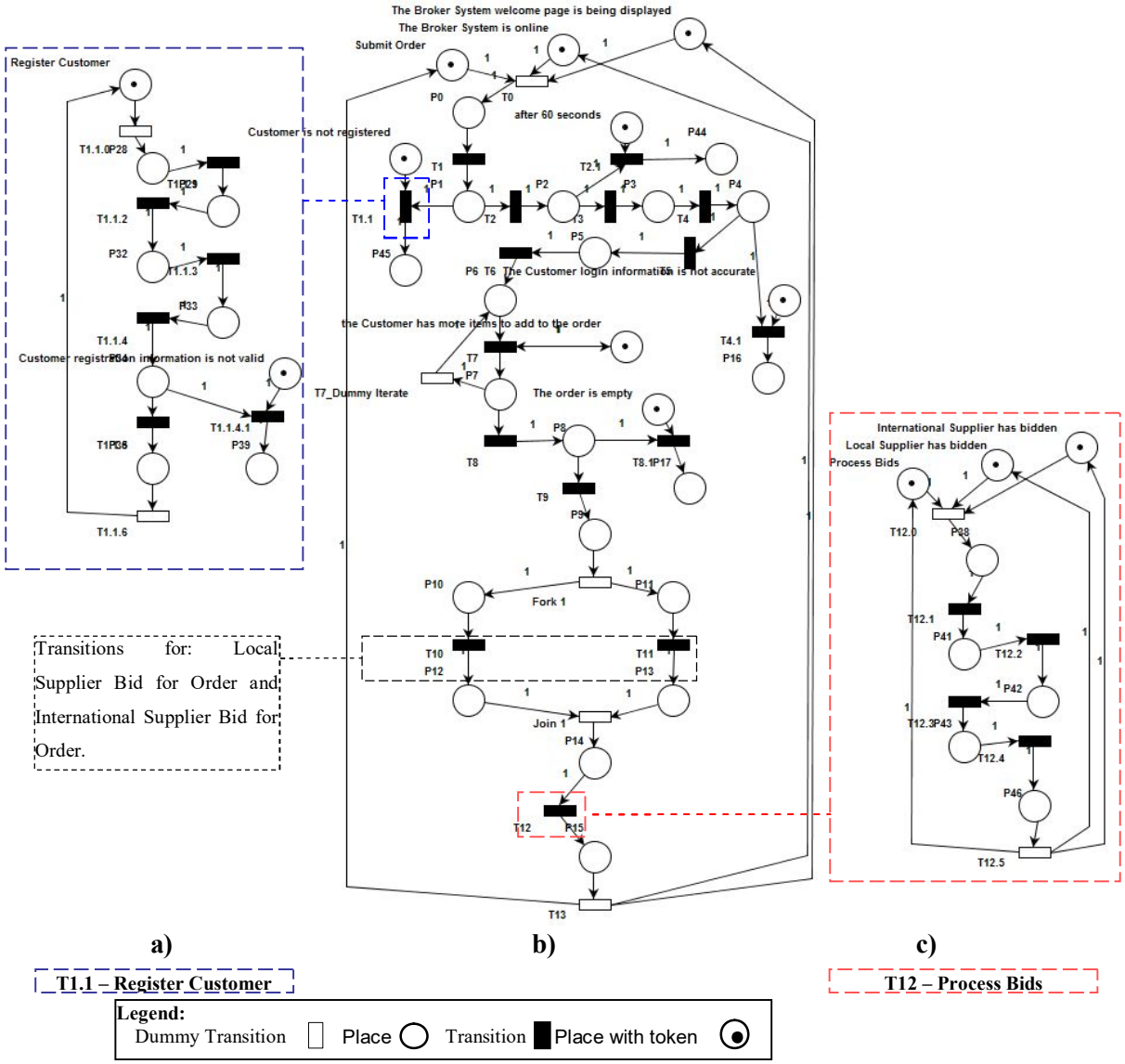
Figure 27 – Integrate Petri-Nets (Method 2).

4.3.3. Petri-Net Example

For illustration, we applied the *Methods 1* and *2* (Figure 26 and Figure 27) to obtain the Petri-Nets and Integrated Petri-Nets of the "Online Broker System".

In the "Online Broker System", we choose the "*Submit Order*" scenario as *main scenario* because it does not require any other scenario of the set of scenarios and references to the most of scenarios of the system. By applying the *Method 1* (Figure 26), we obtain the Petri-Net for the "*Submit Order*" scenario. It was derived by mapping the scenario components of the main execution flow – episodes and exceptions. Figure 22 and Figure 23 depict the set of scenarios of the "Online Broker System", and Figure 28 (b) shows the Petri-Net for the "*Submit Order*" scenario.

For "*Submit Order*" scenario (Figure 22 and Figure 23), 16 event occurrences are identified (12 in the main flow – episodes and 4 in the exceptional flows): T1 (The Customer loads the login page), T2 (The Broker System asks for the Customer login information), T3 (The Customer enters her login information), T4 (The Broker System checks the provided login information), T5 (The Broker System displays an order page), T6 (The Customer creates a new Order), T7 (The Customer adds an item to the Order), T8 (The Customer submits the Order), T9 (The Broker System broadcast the Order to the Suppliers), T10 (LOCAL SUPPLIER BID FOR ORDER), T11 (INTERNATIONAL SUPPLIER BID FOR ORDER), T12 (PROCESS BIDS), T1.1 (REGISTER CUSTOMER), T2.1 (The Broker System displays a login timeout page), T4.1 (The Broker System displays an alert message) and T8.1 (The Broker System displays an error message). We construct a Petri-Net by creating transitions T1, T2,… , T11, T12 and T13 to denote these events and appending to each transition input and output places to denote: (1) internal dummy input and output places, or (2) input conditions (exception's cause or episode's condition) and post-conditions. Additionally: (1) two dummy transitions (Fork1 and Join1) are created for synchronization of concurrent transitions T10 and T11; and (2) two dummy transitions are created to denote the scenario triggering (T0) and the scenario completion (T13).



Legends for Transition Labels

T1 The Customer loads the login page)
T2 The Broker System asks for the Customer login information
T3 The Customer enters her login information
T4 The Broker System checks the provided login information
T5 The Broker System displays an order page
T6 The Customer creates a new Order
T7 The Customer adds an item to the Order
T8 The Customer submits the Order
T9 The Broker System broadcast the Order to the Suppliers
T10 LOCAL SUPPLIER BID FOR ORDER
T11 INTERNATIONAL SUPPLIER BID FOR ORDER
T12 PROCESS BIDS
T1.1 REGISTER CUSTOMER
T2.1 The Broker System displays a login timeout page
T4.1 The Broker System displays an alert message
T8.1 The Broker System displays an error message

Legends for Transition Labels

T1.1.1 Customer selects registration operation
T1.1.2 Broker System asks for Customer name, date of birth and address
T1.1.3 Customer enters registration information
T1.1.4 Broker System validates Customer information
T1.1.5 Broker System generate login information for Customer
T1.1.4.1 Broker System displays registration failure page

T12.1 Customer examines the bid
T12.2 Customer signals the system to proceed with bid
T12.3 HANDLE PAYMENT
T12.4 System put an order with the selected bidder

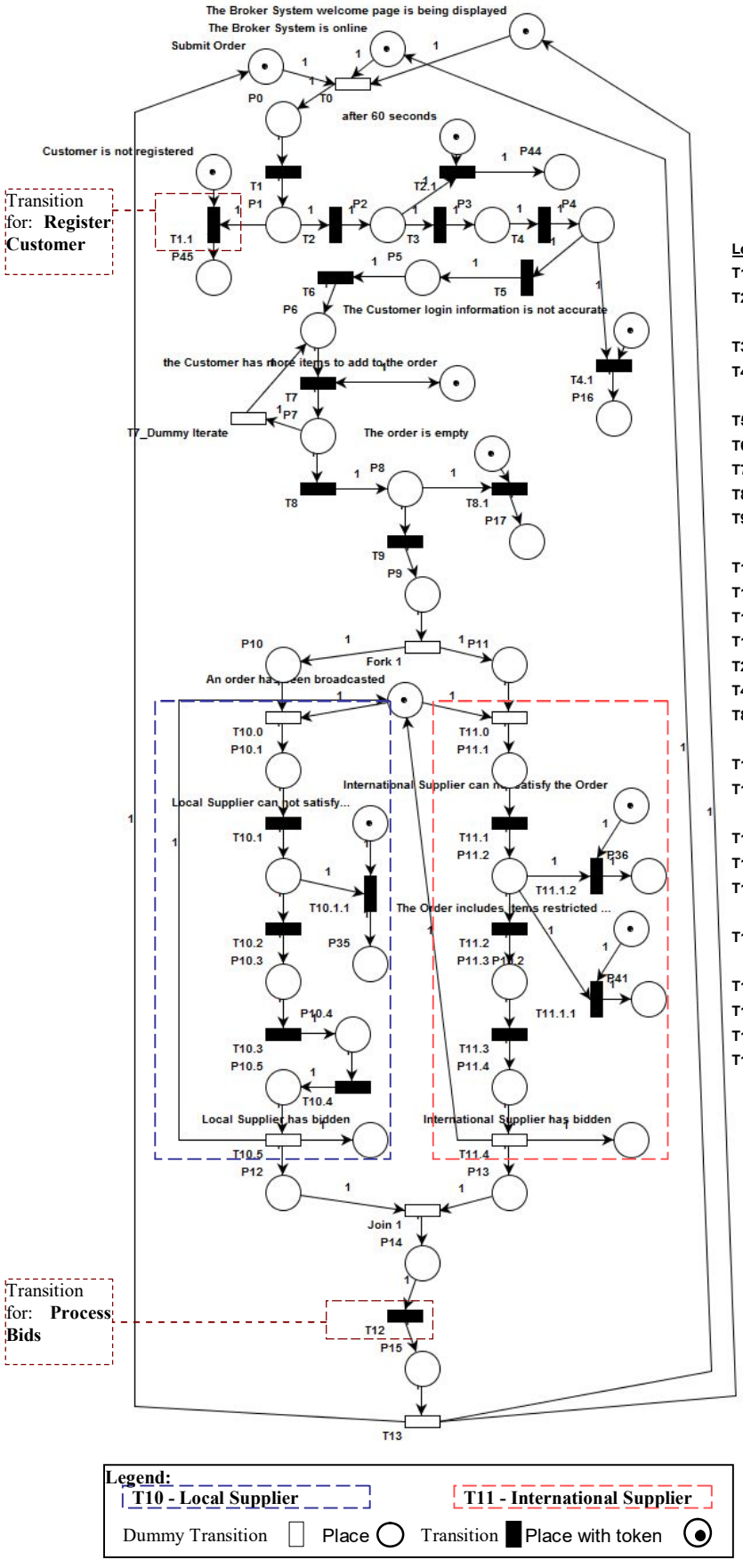
Figure 28 – Register Customer (a), Submit Order (b) and Process Bids (c) Petri-Nets.

Revisiting the "Submit Order" scenario, exception 1.1 and episodes 10, 11 and 12 are detailed in other scenarios (exception and sub-scenario) like "Register

Customer", "Local Supplier bid for order", "International Supplier bid for order" and "Process Bids". It means that Petri-Nets should be generated for referenced scenarios (Register Customer-T1.1, Local Supplier bid for order–T10, International Supplier bid for order–T11 and Process Bids–T12) and replaced into the *main Petri-Net* of "Submit Order".

Figure 28 (a) and (c) show the Petri-Nets derived for like "Register Customer" and "Process Bids" scenarios, and where must be substituted in "Submit Order" Main Petri-Net. In Figure 28 (b), transitions T10 and T11 reference the "Local Supplier bid for order" and "International Supplier bid for order" scenarios; they are executed in a non-sequential order and must be replaced in the corresponding transitions.

Figure **29** shows the Integrated Petri-Net of "Submit Order" scenario. The sequentially related scenarios (T1.1. REGISTER CUSTOMER and T12. PROCESS BIDS) are substituted by the Petri-Nets depicted in Figure 28 (a) and (c).



Legends for Transition Labels

T1 The Customer loads the login page)

T2 The Broker System asks for the Customer login information

T3 The Customer enters her login information

T4 The Broker System checks the provided login information

T5 The Broker System displays an order page

T6 The Customer creates a new Order

T7 The Customer adds an item to the Order

T8 The Customer submits the Order

T9 The Broker System broadcast the Order to the Suppliers

T10 LOCAL SUPPLIER BID FOR ORDER

T11 INTERNATIONAL SUPPLIER BID FOR ORDER

T12 PROCESS BIDS

T1.1 REGISTER CUSTOMER

T2.1 The Broker System displays a login timeout page

T4.1 The Broker System displays an alert message

T8.1 The Broker System displays an error message

T10.1 Local Supplier receives the Order and examines it

T10.2 Local Supplier determines the applicable taxes to the order and creates a bid

T10.3 Local Supplier submits a Bid for the Order

T10.4 The Broker System sends the Bid to the Customer

T10.1.1 Local Supplier passes on the Order

T11.1 International Supplier receives the Order and examines it

T11.2 International Supplier submits a Bid for the Order

T11.3 The Broker System sends the Bid to the Customer

T11.1.1 International Supplier passes on the Order

T11.1.2 International Supplier passes on the Order

Figure 29 - Integrated Petri-Net of "Submit Order".

4.3.4.
Preservation of Properties

We believe that Petri-Net derived from a scenario (***Method 1*** - Figure 26) preserves the event sequences and conditions described within a scenario, as we explain next.

Demonstration 1: A scenario describes situations (Leite et al., 2000) in the form of episodes or exceptions. From a given *initial state* (*context* with all necessary *resources*, *pre-conditions* and *constraints*), the execution of an *episode* or the treatment of an *exception* leads into another *state*. According to the transformation method (***Method 1*** - Figure 26), the execution of a *episode* or the treatment of a *exception* is denoted by the *firing* of a *Petri-Net transition*, which changes a *marking* ***M*** (source state) to ***M'*** (target state). Each *transition* is labeled with the *sentence* or *solution* performed by an episode or exception, respectively. For each *transition* translated from an episode or exception: (1) *Input places* are created to denote the locations of its *pre-conditions, causes* and *constraints;* (2) *Output places* are created to denote the location of its *post-conditions*. Therefore, the execution of scenario episodes or exceptions is modeled by *firing* a sequence of Petri-Net *transitions*.

Moreover, the properties of the Petri-Nets derived from related scenarios are preserved when they are synthesized into a whole Petri-Net, because the synthesis procedure (***Method 2*** - Figure 27) does not introduce new non-deterministic situations (Non-determinism is the main source of synchronization defects), as we explain next.

Demonstration 2: A Petri-Net is the formal representation of a scenario. We integrate the related Petri-Nets in order to obtain a partial initial system design. The integrated Petri-Net reflects exactly the original properties of the synthesized Petri-Nets. Among the synthesized Petri-Nets, there are *common places* (with the same labels) that denote the same *conditions or states*, and there are *places* or *transitions* that reference (in their labels) other Petri-Nets. Our integration method (***Method 2*** - Figure 27) is basically the *fusion of common places* and the *substitution of places or transitions* by the corresponding Petri-Nets: (1) the *substitution* (Definition 6.3, 6.4 and 6.5) of places or transitions by sequentially related Petri-Nets do not create any new arcs between these Petri-Nets, i.e. the

substitution of *places* or *transitions* is done by fusing with the first "*input dummy place (Start)*" or the last "*output dummy place*" (Finish) of the *replacing Petri-Net*; and (2) the *fusion* (Definition 6.6) of concurrently related Petri-Nets does not create any new arcs between these Petri-Nets, i.e. the fusion of places is done by *fusing common places*.

Figure 30 illustrates the application of *substitution input place* and *concurrent fusion place* operations to obtain two integrated Petri-Nets: (a) First example, a *constraint* of a scenario **S1** is detailed in other scenario **S2** (sequential relationship); and (b) Second example, two scenarios **S1** and **S2** interact concurrently because the post-condition of the first one **S1** has the same label that the pre-condition of the second one **S2**.

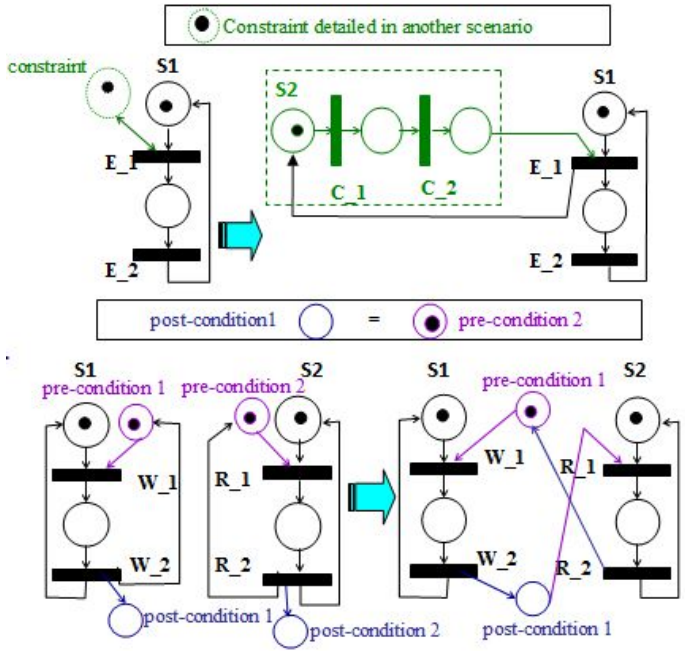


Figure 30 - Substitution input place (a) and concurrent fusion place (b).

4.4. Analyzing Scenarios

The process of analysis involves checking some structural and behavioral properties in *Scenario* descriptions and equivalent *Petri-Nets*, respectively. The analysis of these properties can be addressed through the use of static and dynamic analysis techniques, or the combination of them.

In order to detect defects related to *Unambiguity* and *Completeness*, our analysis approach performs a static analysis. *Unambiguity* analysis detects ambiguous terms or phrases within internal scenario sentences. *Completeness* analysis checks the style and content of internal scenario elements, and detects

defects in the relationships among related scenarios. Some tasks related to these activities can be supported by NLP techniques.

In order to detect defects related to behavioral properties, our analysis approach performs a dynamic analysis. The analysis of the behavior of a set of scenarios can detect inconsistency or incorrectness indicators, such as deadlock situations. Consistency analysis detects some defects due to non-determinism and synchronization issues, we have made use of Place-Transition Petri-Nets (Murata, 1989) for analysis of: (1) static properties like *Correct Token Passing* and *Fully Connected* (related to *Feasibility*); and (2) dynamic properties like *Determinism*, *Boundedness*, *Reversibility* and *Deadlock free* (related to *Non-interferential*, *Boundedness*, *Reversibility* and *Liveness*).

A Quality Model for Scenarios and heuristics to detect defect indicators that hurt *Unambiguity*, *Completeness and Consistency* in scenarios is presented in Chapter 3. Below we detail the steps for *Scenarios* analysis.

4.4.1. Unambiguity Analysis

Natural language plays an important role in scenario specifications because scenario elements are described using NL. Due to the inherent ambiguity, the use of NL is a critical issue. NLP techniques can be used for the linguistic analysis of NL scenario descriptions and search defect indicators that hurt ***Unambiguity***. These indicators can be grouped in categories (*Vagueness*, *Subjectiveness*, *Optionality, Multiplicity, Quantifiability, Readabiity, Minimality, Weakness and Implicitly*) and detected by lexical analysis. *Readabiity* and *Minimality* contributes positively to ***Unambiguity***.

To evaluate ***Readability***, we use the Coleman-Liau Formula readability metric. This metric is based on the number of the letters, words and sentences of a requirement (Wilson et al., 1997).

Coleman-Liau Formula readability metric: (5.89* letters/words-0.3*sentences/(100*words)-15.8]). The reference value of this formula for an easy-to-read technical document is 27.60, if it is < 17.10 and > 55.80 the document is difficult-to-read.

In scenario, the evaluation of properties related to ***unambiguity*** is performed by reading the different scenario elements (typically involving events: title, goal,

episodes and exceptions), and searching for defect indicators (stored in indicators dictionaries) that contribute (positively or negatively) to Unambiguity: *Vagueness*, *Subjectiveness*, *Optionality, Multiplicity, Quantifiability, Minimality, Weakness and Implicitly* (See Table 7, Chapter 3).

A summary of ***unambiguity*** evaluation of *Scenario* elements is outlined below in **Method 3 (Figure 31)**. The detected defects are classified as *Warning* (See Section 4.5).

Method 3: Analyze Unambiguity
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Ambiguous Indicators **A** = {*Vagueness*, *Subjectiveness*, *Optionality, Multiplicity, Quantifiability, Weakness, Implicitly* };
Output: Feedback **F** = (**I**nformations, **W**arnings, **E**rrors)
Begin:
1. Evaluate *Readability* index:
→**IF** Readability Index of Title > 55.8 **THEN** Add "Readability: Title is difficult-to-read" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Readability Index of Sentence > 55.8 **THEN** Add "Readability: Episode is difficult-to-read" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Readability Index of Solution > 55.8 **THEN** Add "Readability: Exception is difficult-to-read" to **W**;
2. Evaluate *Minimality*:
→**IF** Title contains a Text after a not minimal term **THEN** Add "*Minimality*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a Text after a not minimal term **THEN** Add "Minimality: Episode describes a non-minimal sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a Text after a not minimal term **THEN** Add "Minimality: Exception describes a non-minimal solution" to **W**;
3. Evaluate *Vagueness*:
→**IF** Title contains a *Vague* term **THEN** Add "*Vagueness*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Vague* term **THEN** Add "*Vagueness*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Vague* term **THEN** Add "*Vagueness*: Exception describes an ambiguous solution" to **W**;
4. Evaluate *Subjectiveness*:
→**IF** Title contains a *Subjective* term **THEN** Add "*Subjectiveness*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Subjective* term **THEN** Add "*Subjectiveness*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Subjective* term **THEN** Add "*Subjectiveness*: Exception describes an ambiguous solution" to **W**;
5. Evaluate *Optionality*:
→**IF** Title contains a *Optional* term **THEN** Add "*Optionality*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Optional* term **THEN** Add "*Optionality*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Optional* term **THEN** Add "*Optionality*: Exception describes an ambiguous solution" to **W**;
6. Evaluate *Multiplicity*:
→**IF** Title contains a *Multiple* term **THEN** Add "*Multiplicity*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Multiple* term **THEN** Add "*Multiplicity*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Multiple* term **THEN** Add "*Multiplicity*: Exception describes an ambiguous solution" to **W**;
7. Evaluate *Quantifiability*:
→**IF** Title contains a *Quantifiable* term **THEN** Add "*Quantifiability*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Quantifiable* term **THEN** Add "*Quantifiability*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Quantifiable* term **THEN** Add "*Quantifiability*: Exception describes an ambiguous solution" to **W**;
8. Evaluate *Weakness*:
→**IF** Title contains a *Weak* term **THEN** Add "*Weakness*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains a *Weak* term **THEN** Add "*Weakness*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Weak* term **THEN** Add "*Weakness*: Exception describes an ambiguous solution" to **W**;
9. Evaluate *Implicitly*:
→**IF** Title contains an *Implicit* term **THEN** Add "*Implicitly*: Title describes an ambiguous situation" to **W**;
→**For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Sentence contains an *Implicit* term **THEN** Add "*Implicitly*: Episode describes an ambiguous sentence" to **W**;
→**For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Solution contains a *Implicit* term **THEN** Add "*Implicitly*: Exception describes an ambiguous solution" to **W**;
10. Return Feedback **F =** {**W**};
End

Figure 31 – Unambiguity Analysis (Method 3).

Following are examples of *Unambiguity* defects pointed out by our analysis approach in the "Online Broker System"; the underlined words or phrases are the indicators detected by our approach to point out the episode sentence containing the defect:

- **Submit Order scenario**: Episode 1. The Customer enters **her** login information (*Implicitly*);
- **Submit Order scenario**: Episode 4. The Broker System checks the **provided** login information (*Vagueness*);
- **Local Supplier scenario:** Episode 3. Local Supplier receives the Order **and** examines it (*Multiplicity*);
- **Handle Payment:** Episode 2. The Customer provides **her** Credit Card information (*Implicitly*).

4.4.2. Completeness Analysis

To evaluate ***Completeness***, we detect missing information in internal (intra-scenario) and external aspects (inter-scenario) of scenarios. The intra-scenario properties include: ***Atomicity***, ***Simplicity***, ***Uniformity***, ***Usefulness*** and ***Conceptually Soundness***. The inter-scenario properties include: ***Integrity***, ***Coherency*** and ***Uniqueness***. Other important property related to ***completeness*** is ***Feasibility*** (See Chapter 3).

Thus, the syntax and semantic of each element in scenario and its relationships must be described as established in the scenario model and grammar (Table 13).

The violation of properties related to completeness is detected by traversing every scenario element (Title, Goal, Context, Resource, Actor, Episodes, Exception), and following the checklist with verification heuristics described in Chapter 3 (Table 8, Table 9, Table 10 and Table 11). For each heuristic, we defined a set of common defect indicators.

In order to search defect indicators that hurt *Completeness* properties, we classify the defects detection heuristics according to the analysis strategy used by them, i.e., some of the heuristics verify that every scenario element contains its relevant components (*Lexical*), others verify that every scenario elements and its

internal components follows the grammar rules (*Syntactic*). In Chapter 5 is detailed the classification of each one of the defects detection heuristics.

4.4.2.1. Lexical Analysis

To detect *lexical* defects that hurt *Uniformity* it is enough to verify the conformance of scenario elements to the scenario model. Figure 32 illustrates examples of lexical analysis of an episode (a) and an exception (a) described in the "Online Broker System" scenarios.

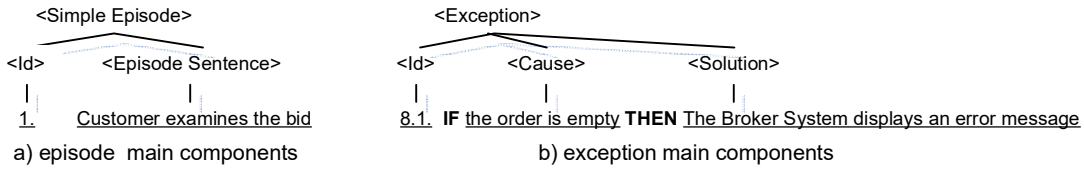


Figure 32 – Lexical analysis of simple episode (a) and exception (b) elements.

To detect *lexical* defects that hurt *Atomicity, Simplicity, Usefulness, Conceptually Soundness, Integrity* and *Coherency*, it is enough to search for multiplicity indicators in the scenario title, to count the number of episodes in each scenario, to check that every actor or resource is used in episodes, to verify the presence of *Linking-verb* or *State-Verb* in conditions (pre-condition, post-condition, episode condition and exception cause), to verify the existence of referenced scenarios or pre-conditions, and to check the coherency between related scenario pre-conditions (geographical location and temporal location), respectively.

Following are examples of *Completeness lexical* defects pointed out by our analysis approach in the "Online Broker System"; the underlined words or phrases are the indicators detected by our approach to point out the sentence containing the defect:

- **International Supplier bid for order**: Exception 1.1 IF The Order includes items restricted for exportation THEN International Supplier passes on the Order (*Soundness* - Missing Linking-Verb or State-Verb);
- **Submit Order**: Num. episodes > 10 (*Usefulness* **-** Too long scenario);
- **Submit Order**: Context Pre-condition - The Broker System is online (*Integrity* – It is an uncontrollable fact does not satisfied by a Post-condition of other scenario);

- **Process Bids**: Actor – Broker System (*Usefulness* - never participates in episodes).

4.4.2.2. Syntactical Analysis

To detect *syntactic* defects that hurt *Atomicity, Simplicity, Usefulness, Conceptually Soundness* and *Uniqueness*, it is necessary to check that every sentence contains significant information like the main *Verb*, Direct *Object* modified, and optionally the *Subject* and *Indirect Objects*. Considering that a scenario sentence (typically involving events: title, episode sentence and exception solution) performs an action (Action-Verb) that can use or modify resources (Objects) and be executed by actors (Subjects), there are three basic types of structured sentences: 1) *verb-object* (for writing the scenario title or reference another scenario), 2) *subject-verb-object*, and 3) *subject-verb-object-indirect-object* (for writing episode sentences or exception solutions).

Phrase-structure parsing or dependency parsing (Stanford, 2015) strategies can be used to identify the significant information of scenario sentences. The result of the parsing is a parse tree, in which the sentence is parsed into the *Subject* or *Object* of a *Verb*; then the non-leaf nodes are Part-of-Speech (Klein and Manning, 2003) tags where "NN" and "VB" represents the noun phrase and verb phrase respectively; the leaf nodes are tokenized words of the original textual sentence. *Stanford* (2015) tool is a program that could be used to analyze the grammatical structure of sentences.

Figure 33 illustrates the parse tree for sentences described in the "Online Broker System" scenarios.

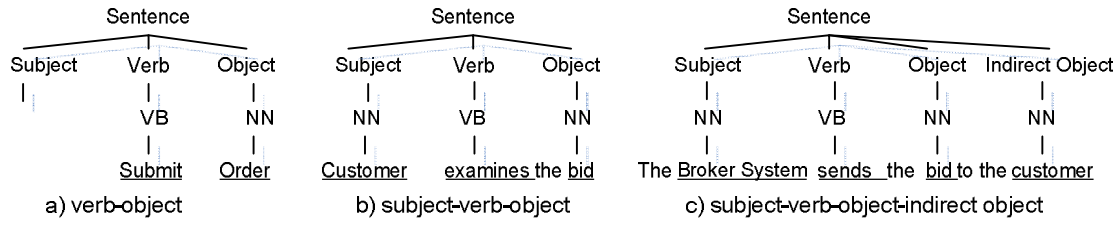


Figure 33 – Parse tree for verb-object (a), subject-verb-object (b) and subject-verb-object-indirect-object (c) sentences.

Following are examples of *Completeness syntactic* defects pointed out by our analysis approach in the "Online Broker System"; the underlined words or

phrases are the indicators detected by our approach to point out the sentence containing the defect:

- **Local Supplier bid for order**: Episode 1. Local Supplier receives the Order and examines it (*Simplicity* - Contains more than one Action-Verb);
- **International Supplier bid for order**: Episode 3. The Broker System sends the Bid to the Customer (*Usefulness* – Actor/Resource mentioned in episode is not included in the Actor/Resource element);
- **Process Bids**: Episode 4. System put an order with the selected bidder (*Simplicity* – Missing Action-Verb in Present Tense form);
- **Local Supplier bid for order**: "Local Supplier bid for order" and "International Supplier bid for order" (*Uniqueness* – They are potentially duplicated! Their Titles share the same Action-Verb and direct Object).

Method 4: Analyze Completeness
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Completeness Defect Indicators **C** = {*Atomicity, Simplicity, Uniformity, Usefulness*, *Conceptually Soundness, Integrity, Coherency, Uniqueness, Feasibility*};
Output: Feedback **F** = (**I**nformations, **W**arnings, **E**rrors)
Begin:
1. **IF** Title describes multiple situations **THEN** Add "Atomicity: Title must express only a situation" to **W**;
2. **IF** Goal describes multiple situations **THEN** Add "Atomicity: Goal must express only a situation" to **W**;
3. For each Scenario Element in Scenario **S**:
→**IF** Scenario Element does not follow the scenario model **THEN** Add "Uniformity: Scenario Element must follow the scenario model" to **E**;
4. **For** every ***Episode*** in episodes of Scenario **S**:
→**IF** Episode is not *Readable* **THEN** Add "Simplicity: Episode is difficult-to-read" to **W**;
→**IF** Episode is not consistent with Actors and Resources **THEN** Add "Usefulness: Episode must be consistent with actors and resources" to **W**;
→**IF** Episode does not perform actions and change states **THEN** Add "Conceptually Soundness: Episode should perform actions and change states" to **W**;
5. **For** every ***Exception*** in exceptions of Scenario **S**:
→**IF** Exception is not *Readable* **THEN** Add "Simplicity: Exception is difficult-to-read" to **W**;
→**IF** Exception is not consistent with episodes **THEN** Add "Usefulness: Exception must be consistent with branching episode" to **W**;
→**IF** Exception does not perform actions and change states **THEN** Add "Conceptually Soundness: Exception should perform actions and change states" to **W**;
6. Identify sequential relationships of the Scenario **S** by Pre-condition. Post-condition, Constraint, Sub-scenario or Exception;
7. **For** every Related Scenario in Sequentially Scenarios:
→**IF** Related Scenario does not exist in the set of scenarios **THEN** Add "Integrity: Related Scenario should exist in the set of scenarios" to **E**;
→**IF** Related Scenario does not use a common terminology with the Scenario **S THEN** Add "Coherency: Related Scenario should use a common terminology, e.g. pre-conditions, temporal location and geographical location should be coherent with the main Scenario" to **I**;
8. **Get** the set of Scenarios of the Project;
9. **For** every Scenario **SS** in the set of Scenarios of the Project:
→**IF** Scenario **SS** is duplicated of Scenario **S THEN** Add "Uniqueness: scenarios should not share the same *Title*, *Goal* or *Episodes*" to **W**;
10. Derive Petri-Net **PN** from Scenario **S** (Method 1);
11. **IF** there are places or transitions that do not interact with others in **PN THEN** Add "Feasibility: Petri-Net contains isolated sub nets" to **E**;
12. Return Feedback **F**;
End

Figure 34 – Completeness Analysis (Method 4).

A summary of ***Completeness*** evaluation of a *Scenario* and its related scenarios is outlined in **Method 4 (Figure 34)**. The detected defects are classified as *Information*, *Warning* or *Error* (See Section 4.5).

4.4.3. Consistency Analysis

According to Denger et al (2005), the most difficult defects to detect by static analysis are *consistency* defects; these defects can be detected with much effort using reading or inspection techniques. In order to address this issue, we integrated the dynamic analysis.

Dynamic analysis of scenarios can be performed by rigorous analysis techniques, i.e., from a *main scenario*, a set of *related scenarios* (sequentially and non-sequentially related) are identified and translated into *executable models* (Petri-Nets), which are executed in a formal analysis environment like PIPE2 (2015).

To evaluate ***Consistency***, we integrate the Petri-Nets corresponding to related scenarios into the Petri-Net derived from a *main scenario*, and detect wrong information in the *Integrated Petri-Net*. The *consistency* related properties include: ***Non-interferential***, ***Boundedness***, ***Reversibility*** and ***Liveness*** (See Chapter 3).

The violation of properties related to ***consistency*** can be detected by generating the reachability graph of the equivalent *Integrated Petri-Net*, and analyzing this graph following the checklist with verification heuristics described in Chapter 3 (Table 12). For each heuristic, we identified a set of common defect indicators.

4.4.3.1. Managing the State Explosion

State explosion issue is a serious problem when applying *Petri-Net analysis* to large systems. A contribution of this thesis is a ***MULTI-STEP consistency analysis*** method to manage this problem. The reachability analysis of an *Integrated Petri-Net* can be performed in a compositional way, where: (1) *Petri-Nets* corresponding to *sequentially related scenarios* are removed from the *Integrated Petri-Net*, (2) *Petri-Nets* corresponding to *non-sequentially related*

scenarios are preserved into the *Integrated Petri-Net* because they might interact among them, and (3) the resulting Petri-Nets are analyzed separately.

In this method, the *Integrated Petri-Net* is divided into a set of Petri-Nets (Petri-Nets corresponding to sequentially related scenarios and the *Integrated Petri-Net*) that preserves the properties and concurrency characteristics of the *Integrated Petri-Net*.

It is possible because the process to obtain an *Integrated Petri-Net* from a *main scenario* and its relationships does not introduce new arcs when a Petri-Net corresponding to a related scenario is fused or substituted into a place or transition of the *Integrated Petri-Net* (Method 2 - Figure 27, Section 4.3.2).

A summary of the MULTI-STEP ***Consistency*** evaluation of an equivalent Petri-Net is outlined below in **Method 5** (Figure 35). The detected defects are classified as *Information*, *Warning* or *Error* (See Section 4.5).

Method 5: Analyze Consistency
Input: Scenario ***S*** = (Title, Goal, Context, Resource, Actor, Episodes, Exception);
Consistency Defect Indicators **C** = *{Non-interferential, Boundedness, Reversibility, Liveness };*
Output: Feedback **F** = (**I**nformations, **W**arnings, **E**rrors)
Begin:
1. Derive the Main Petri-Net **RPN** from the *Root Scenario* **S** (Method 1);
2. Identify *sequential relationships* from the Main ***Scenario*** by *Pre-condition. Post-condition, Constraint, Sub-scenario or Exception* (Section 4.1);
3. Identify explicit *non-sequential relationships* from the Main ***Scenario*** by analyzing Concurrency Constructs (Section 4.1);
4. Identify non-explicit *non-sequential relationships* from the Main ***Scenario*** by common *Pre-condition* or *Post-condition* (Section 4.1);
5. For every Sequentially Related Scenario:
→Derive Petri-Net **PN** (Method 1);
→Add the Petri-Net **PN** into a Set of Petri-Nets **SPN**;
6. For every Non-Sequentially Related Scenario:
→Derive Petri-Net **PN** (Method 1);
→Integrate the Petri-Net **PN** into the Main Petri-Net **RPN** (Method 2);
7. Add the Integrated Petri-Net **RPN** into the Set of Petri-Nets **SPN**;
8. For every Petri-Net **PN** in the Set of Petri-Nets **SPN**:
8.1. Generate the Reachability Graph of the Petri-Net **PN**;
8.2. Analyze the Reachability Graph of the Petri-Net **PN**:
→**IF** Petri-Net **PN** contains simultaneously enabled transitions **THEN** Add "*Non-interferential*: Contains simultaneously enabled transitions" to **I**;
→**IF** Petri-Net **PN** contains overflowed places **THEN** Add "*Boundedness*: The number of elements in a place exceeds a finite capacity" to **W**;
→ **IF** Petri-Net **PN** is not reversible **THEN** Add "*Reversibility*: Error recovery is not possible" to **W**;
→ **IF** Petri-Net **PN** contains a path to deadlock **THEN** Add "*Liveness*: Exist a short path to deadlock" to **W**;
→ **IF** Petri-Net **PN** contains never enabled transitions **THEN** Add "*Liveness*: Exist not enabled transitions" to **W**;
9. Return Feedback **F**;
End

Figure 35 – Consistency Analysis (Methoid 5).

This MULTI-STEP method reduces the state explosion problem by: (1) increasing the feasibility of Petri-Nets and (2) enabling the verification of

properties which may fail on *Integrated Petri-Net* due to a state explosion problem.

In the *"Online Broker System",* the main scenario is the "Submit Order" scenario, and the Suppliers' scenarios are executed concurrently (non-sequential), as shown in Figure 22 and Figure 23.

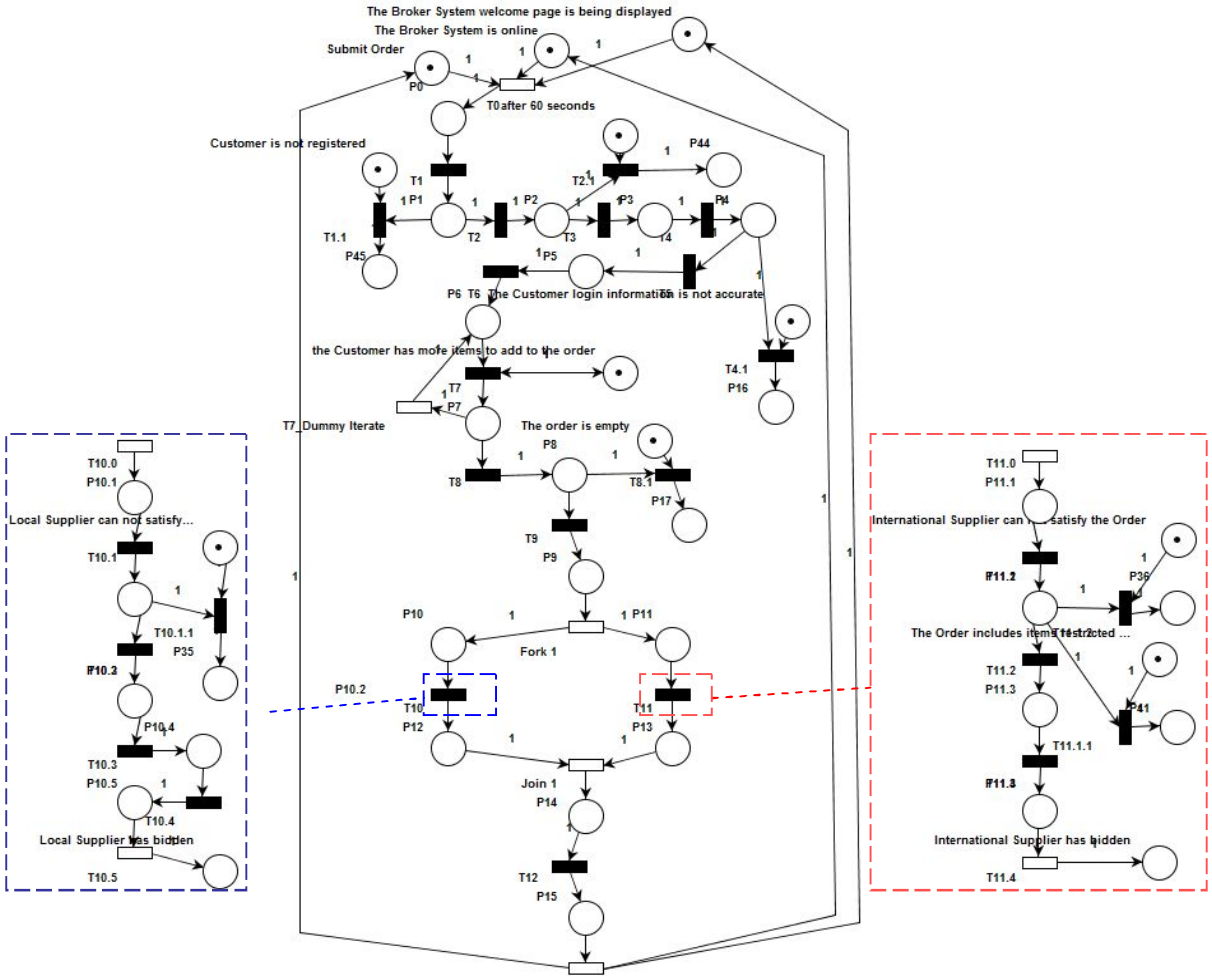
Figure **36** depicts how the Petri-Nets derived from non-sequentially related scenarios (LOCAL SUPPLIER BID FOR ORDER and INTERNATIONAL SUPPLIER BID FOR ORDER) are substituted into the corresponding transitions (T10 and T11) of the *Petri-Net* derived from the "Submit Order" scenario. The Petri-Nets corresponding to *REGISTER CUSTOMER and PROCESS BIDS* transitions (T1.1 and T12) are not integrated because they are sequentially related to "Submit Order" scenario, and they can be analyzed separately because they do not interact concurrently with the main scenario (avoiding the state explosion issue).

From the *Integrated Petri-Net* in **Figure 36**, we: a) generate the reachability graph, and b) apply the reachability analysis technique to detect consistency defects. **Figure 37** depicts the reachability graph and results of the reachability analysis for "Submit Order" scenario using the PIPE2 (2015). Nodes are reachable states; arcs are transitions performed to reach a state, and S_0 is the initial state.

The reachability analysis of the *integrated Petri-Net* PN of "Submit Order" scenario (Figure 22 and Figure 23) using the PIPE2 (2015) tool pointed out the following *Consistency* defects in the "Online Broker System":

- Non ***bounded*** because it presents overflowed places (Local Supplier has bidden, International Supplier has bidden);
- Non ***live*** because the firing sequence ⟨(SUBMIT ORDER) T0 → T1 → T2 → T3 → T4 → T5 → T6 → T7 → T8 → T9 → Fork_1 → (LOCAL SUPPLIER BID FOR ORDER) T10.0 → T10.1 → T10.2 → T10.3 → T10.4 → T10.5 → (INTERNATIONAL SUPPLIER BID FOR ORDER) T11.0 → T11.1 → T11.2 → T11.3 → T11.4 → (SUBMIT ORDER) Join_1 → T12 → T12 → T13 → T0 → T1 → T2 → T3 → T4 → T5 → T6 → T7 → T8 → T9 → Fork_1 → (LOCAL SUPPLIER BID FOR ORDER) T10.0 → T10.1 → T10.2 → T10.3 → T10.4 → T10.5 → (INTERNATIONAL SUPPLIER BID FOR ORDER) T11.0 → T11.1 → **T11.1.1**⟩ is a shortest path to Deadlock;

- Non ***reversible***, because it is not *bounded*, not *safe* and not *live.* There is a deadlock when the "*The Order includes items restricted for exportation*" in the "International Supplier Bid for Order" scenario.



Legends for Transition Labels

T1 The Customer loads the login page)
T2 The Broker System asks for the Customer login information
T3 The Customer enters her login information
T4 The Broker System checks the provided login information
T5 The Broker System displays an order page
T6 The Customer creates a new Order
T7 The Customer adds an item to the Order
T8 The Customer submits the Order
T9 The Broker System broadcast the Order to the Suppliers
T10 LOCAL SUPPLIER BID FOR ORDER
T11 INTERNATIONAL SUPPLIER BID FOR ORDER
T12 PROCESS BIDS
T1.1 REGISTER CUSTOMER
T2.1 The Broker System displays a login timeout page
T4.1 The Broker System displays an alert message
T8.1 The Broker System displays an error message

Legends for Transition Labels

T10.1 Local Supplier receives the Order and examines it
T10.2 Local Supplier determines the applicable taxes to the order and creates a bid
T10.3 Local Supplier submits a Bid for the Order
T10.4 The Broker System sends the Bid to the Customer
T10.1.1 Local Supplier passes on the Order

T11.1 International Supplier receives the Order and examines it
T11.2 International Supplier submits a Bid for the Order
T11.3 The Broker System sends the Bid to the Customer
T11.1.1 International Supplier passes on the Order
T11.1.2 International Supplier passes on the Order

P<x><y> are input or output dummy places

Figure 36 – Integrating "Suppliers" Petri-Nets into the Petri-Net of "Submit Order".

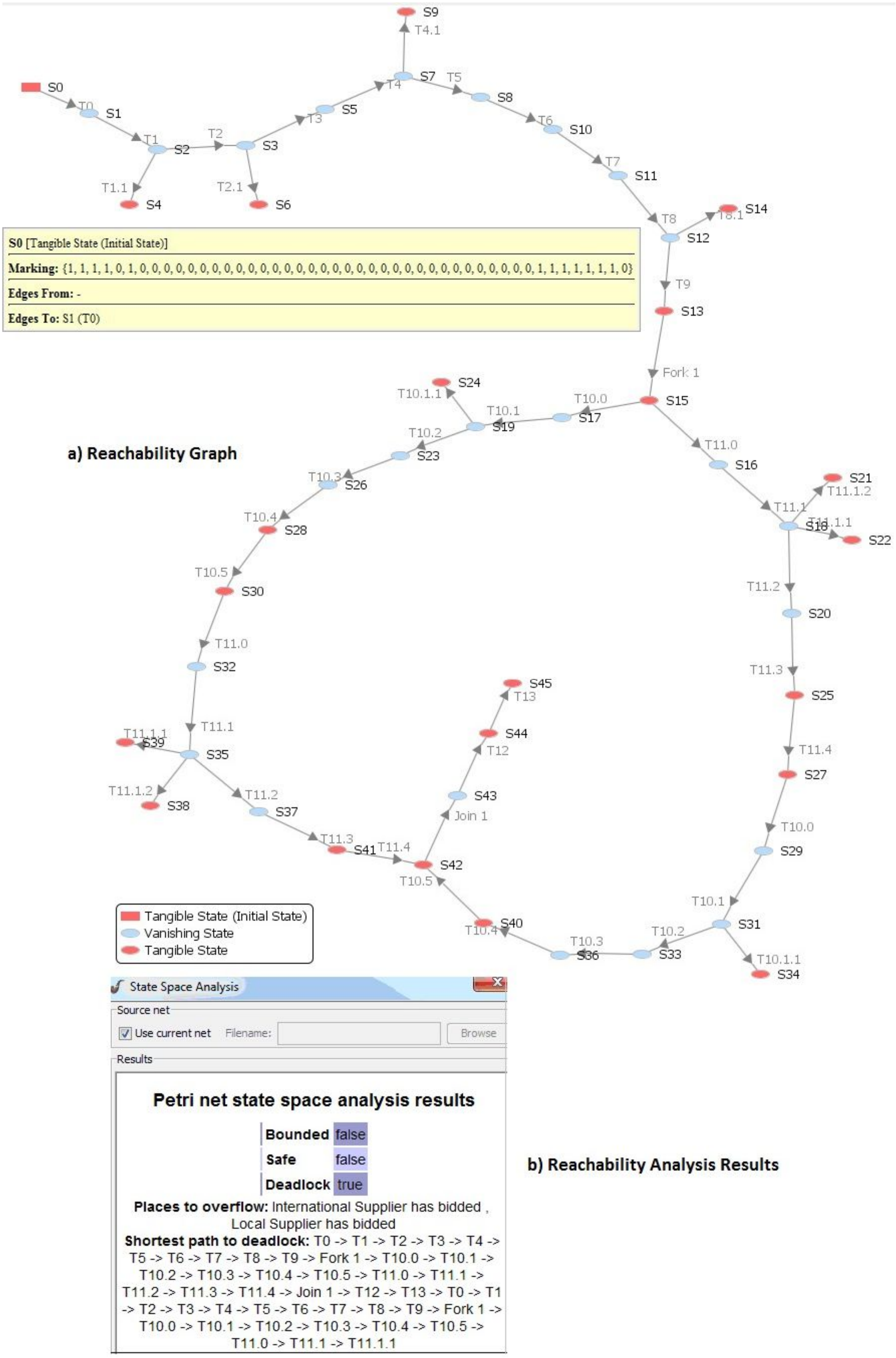


Figure 37 – Reachability graph (a) and Reachability analysis results (b) of “Submit Order” scenario.

4.4.4. Correctness Analysis

Correctness is the main quality in scenarios, and it is difficult to evaluate and achieve because it depends on semantic analysis of scenarios and user's satisfaction. To address some issues related to the *correctness* of scenarios, we could formalize the scenario descriptions using formal methods or analyze semantically the information contained in sentences described within scenarios.

Formal methods are a powerful means to evaluate scenarios because they provide a theoretical framework in which erroneous situations could be predicted. However, specific skills are needed, and this increases their application cost.

The use of NLP techniques could help in identifying the syntactic information of sentences, i.e. NLP techniques can identify Part-of-Speech (POS) tags like "Nouns" and "Verbs" in textual sentences. However, it is impossible to detect semantic defects with high precision (Lucassen et al., 2015).

In this work, NLP techniques (syntactic analysis) and Formal methods (Petri-Nets) are used for detecting defects that hurt *Completeness* and *Consistency* qualities, respectively. The use of these techniques can contribute positively to the *Correctness* of scenarios.

In our analysis approach, we introduced a novel perception of *correctness* and its complex relationships with *unambiguity*, *completeness* and *consistency* describing it as a quality that should be satisficed by contributions of related qualities or properties (See Chapter 3).

4.5. Generating Feedback

By combining static and dynamic analysis techniques, we are able to detect defects that hurt the properties related to *unambiguity*, *completeness* and *consistency* qualities, and, consequently address the defects that hurt *Correctness* of scenario-based specifications.

In Table 20, we summarize how the defects are detected and classified by our analysis approach (Method 3, 4 and 5). These defects are detected by heuristics that implement *Lexical*, *Syntactical* or *Reachability* analysis strategies and classified as: *Information*, *Warning* or *Error*. Implementation details of these heuristics are presented in Chapter 5 (Section 5.3.5). *Information* reveals that the

requirements engineer may have forgotten to specify some information related to a scenario element. *Warning* reveals that the requirements engineer may have introduced some confusing information or forgotten to inform and important scenario element. *Error* reveals that the requirements engineer may have introduced wrong information related to a scenario element.

Table 20 - Scenario Defects Classification

Quality	Property	Heuristic	Analysis Strategy	Defect Category
Unambiguity	***Vagueness***	1	Lexical	Warning
	Subjectiveness	1	Lexical	Warning
	Optionality	1	Lexical	Warning
	Weakness	1	Lexical	Warning
	Multiplicity	1	Lexical	Warning
	Implicitly	1	Lexical	Warning
	Quantifiability	1	Lexical	Warning
	Readability	1	Lexical	Warning
Completeness	***Atomicity***	1	Lexical	Warning
		2	Lexical	Warning
		3	Syntactic	Warning
	Simplicity	1	Syntactic	Warning
		2	Syntactic	Warning
		3	Lexical	Information
		4	Lexical	Warning
		5	Lexical	Warning
		6	Lexical	Warning
	Uniformity	1	Lexical	Warning
	Usefulness	1	Lexical	Warning
		2	Syntactic	Warning
		3	Lexical	Warning
		4	Syntactic	Warning
		5	Lexical	Warning
		6	Lexical	Warning
	Conceptually Soundness	1	Syntactic	Warning
		2	Semantic	Warning
		3	Semantic	Warning
		4	Syntactic	Warning
		5	Lexical	Information
		6	Lexical	Information
		7	Lexical	Information
		8	Syntactic	Warning
		9	Lexical	Information
	Integrity	1	Lexical	Error
		2	Lexical	Information
		3	Lexical	Information
	Coherency	1	Semantic	Warning
		2	Lexical	Warning
		3	Lexical	Warning
	Uniqueness	1	Lexical	Warning
		2	Lexical	Warning
		3	Lexical	Warning
		4	Lexical	Warning
		5	Syntactic	Warning
	Feasibility	1	Lexical	Error
		2	Lexical	Error
Consisyency	***Non-interferential***	1	Reachability analysis	Information
	Boundedness	1	Reachability analysis	Warning
	Reversibility	1	Reachability analysis	Warning
	Liveness	1	Reachability analysis	Warning

The presence of defects classified as *Information*, *Warning* or *Error* is likely, although not conclusively, to be incorrect and must be fixed. Some of these defects can have been introduced on purpose by requirements engineers and that the final decision can be made only in the next software development activities.

4.5.1. Traceability between Petri-Net and Scenario

Every *transition* in a Petri-Net denotes an event occurrence (episode sentence or exception solution) in its corresponding scenario. Every *place* denotes the location of a *pre-condition, post-condition, cause* or *constraint*. If event labels, condition labels and constraint labels are assigned to these Petri-Net *transitions* and *places* accordingly; then, defects in Petri-Net can be translated to defects in scenario:

- *Non-interferential*: Simultaneous enabled transitions represent simultaneous enabled episode sentences or exception solutions;
- *Boundedness*: An overflowed place represents a *pre-condition, post-condition, cause* or *constraint*;
- *Liveness*: A path to deadlock represents an ordered sequence of episode sentences or exception solutions from scenario initial state;

4.6. Recommending Fixes for Defects

The final activity is that of giving advice to requirements engineers about the defect detected by our scenario analysis approach. Given the detailed information provided by our scenario analysis approach (e.g., it indicates the source of the defect), we have developed a rule-based heuristic as part of our scenario analysis approach in order to recommend fixes to requirements engineers, so that they can review scenario descriptions and deal with defects that hurt the properties related to *unambiguity*, *completeness* and *consistency* via refactoring of scenarios. A similar strategy based on recommendation tables was proposed by Rago et al. (2014).

Below, we list some general heuristics for generating useful recommendations. Table 21, Table 22, Table 23 and Table 24 show the recommendation to be provided by our analysis approach if some defect indicator

is found within internal scenario elements or scenario's relationships. The last three columns of the table contain the property evaluation heuristic, defect indicators detected by the heuristic, and the recommendation given to the requirements engineers under the given defect indicator. It is up to the requirements engineers to decide whether a defect is correctly detected with our automated analysis approach and if they should follow the recommendation to fix that defect.

Table 21 – Recommendations for Analyzing Unambiguity Properties.

Property	Heuristic	Indicator	Recommendation
Vagueness	1	A scenario <sentence> contains a vague term.	Remove the vague term
Subjectiveness	1	A scenario <sentence> contains a subjective term.	Remove the subjective term
Optionality	1	A scenario <sentence> contains a optional term.	Remove the optional term
Weakness	1	A scenario <sentence> contains a weak term.	Remove the weak term
Multiplicity	1	A scenario <sentence> contains a multiple term.	Split the sentence into multiple sentences
Implicitly	1	A scenario <sentence> contains an implicit term.	Remove the implicit term
Quantifiability	1	A scenario <sentence> contains a quantifiable term.	Remove the quantifiable term
Minimality	1	A scenario <sentence> contains a Text after a not minimal term.	Split the sentence into multiple sentences
Readability	1	A scenario <sentence> is difficult-to-read.	Check that sentence contains significant information like the main verb, direct object and optionally the subject

Table 22 – Recommendations for Analyzing Completeness (Intra-Scenario).

Property	Heuristic	Indicator	Recommendation
Atomicity	1	The scenario Title contains a multiple term.	Remove *and*, *or*, *and/or* terms
	2	The scenario Goal contain a multiple term.	Remove *and*, *or*, *and/or* terms
	3	Missing *Action-Verb* in Title	Inform an *action-verb* in infinitive form
		Missing *Object* in Title	Inform an *object*
Simplicity	1	*Episode/Exception* contains more than one *Action-Verb*	Split the sentence into multiple sentences
		Episode/Exception contains more than one *Subject*	Split the sentence into multiple sentences
		Missing *Subject* in *Episode/Exception*	IF sentence do not reference another scenario THEN inform a *subject*
		Missing *Object* in *Episode/Exception*	Inform an *object*
	2	The *Action-verb* is not in the *third form* in *Episode/Exception*	Use an *action-verb* in the present simple tense and active form
	3	*Title* contains text between brackets	Remove unnecessary information between brackets
	4	Duplicated *Episode Id*	Remove or re-write one episode
		Duplicated *Episode sentence*	Remove or re-write one episode
	5	More than one *Episode* inside a nested *IF*	Extract the sequence to a separate scenario
	6	More than one *Sentence* inside a *Exception Solution*	Extract the sequence to a separate scenario
Uniformity	1	Missing *Title*	Inform the *Title*
		Missing *Goal*	Inform the *Goal*
		Missing *Actors*	Inform the *Actors*
		Missing *Resources*	Inform the *Resources*
		Context does not contain its relevant sub-components	Inform at least a *Pre-condition, Post-condition, Temporal Location or Geographical Location*
		Missing *Episodes*	Inform the *Episodes*
		Episode does not contain its relevant parts (Id, Sentence)	1. IF episode is Conditional or Loop THEN inform at least: Id, Condition and Sentence; 2. IF episode is Simple THEN inform at least: Id and Sentence;
		Exception does not contain its relevant parts (Id, Cause, Solution)	Inform: Id, Cause and Solution

Table 23 – Recommendations for Analyzing Completeness (Intra-Scenario).

Property	Heuristic	Indicator	Recommendation
Usefulness	1	Actor does *not participate* in the situation;	Mention the actor in at least an episode
	2	Missing *Actor* in Actors element;	Include the actor in the Actors
	3	Resource that is *not used* in the situation;	Mention the resource in at least an episode
	4	Missing *Resource* in Resources element;	Include the resource in the Resources
	5	Branching *Episode* of an exception is *missing*;	Update the *exception Id* to appoint the correct episode
	6	Number of *episodes* in each scenario is less than 3 or more than *9*;	Re-write the scenario to keep between 3 and 9 episodes
Conceptually Soundness	1	The corresponding *verbs* and *objects* appearing in the two compared sentences are not the same	Re-write the Title to satisfy the Goal
	2		
	3		
	4	Missing *Action-Verb* in episode sentences;	Inform an *action-verb*
	5	Missing *Linking-Verb* in episode conditions;	Inform a *linking-verb*
	6	Missing *State-Verb* in Pre-conditions;	Inform a *state-verb*
	7	Missing *State-Verb* in Post-conditions;	Inform a *state-verb*
	8	Missing *Action-Verb* in exception solution;	Inform an *action-verb*
	9	Missing *Linking-Verb or State-Verb* in exception causes;	Inform a *linking-verb* or *state-verb*

Table 24 – Recommendations for Analyzing Completeness (Inter-Scenario).

Property	Heuristic	Indicator	Recommendation
Integrity	1	*Pre-condition* identified as related scenario *does not exist* within the set of scenarios;	Include the *related scenario* to the set of scenarios
		Post-condition identified as related scenario *does not exist* within the set of scenarios;	Include the *related scenario* to the set of scenarios
		Episode sentence identified as related scenario *does not exist* within the set of scenarios;	Include the *related scenario* to the set of scenarios
		Exception solution identified as related scenario *does not exist* within the set of scenarios;	Include the *related scenario* to the set of scenarios
		Constraint identified as related scenario *does not exist* within the set of scenarios;	Include the *related scenario* to the set of scenarios
	2	Complex *Exception Solution* must be treated by a scenario;	Include the *exception solution* to the set of scenarios
	3	Missing *pre-condition/post-condition;*	IF the *pre-condition* is not an uncontrollable fact THEN describe it as *post-condition* of another scenario
Coherency	1		
	2	Related scenario Geographical location is not in the set of Geographical locations of root scenario;	Re-write the geographical locations of related scenario to be more restrict to the main scenario.
		Related scenario Temporal location is not in the set of Temporal locations of root scenario;	Re-write the temporal locations of related scenario to be more restrict to the main scenario.
	3	Circular inclusion between two scenarios;	Remove the reference to the main scenario (in referenced scenario)
Uniqueness	1	*Title* coincidence between two scenarios;	1. IF the sets of episodes are the same THEN remove one scenario; 2. IF the sets of episodes are not the same THEN rename the Title of one scenario;
	2	*Goal* coincidence between two scenarios;	1. IF the sets of episodes are the same THEN remove one scenario; 2. IF the sets of episodes are not the same THEN rename the Goal of one scenario;
	3	*Pre-condition* coincidence between two scenarios;	IF the sets of episodes are the same THEN remove one scenario;
	4	*Episodes* coincidence between two scenarios;	1. IF the set of episodes of scenario_2 is included in scenario_1 THEN remove the duplicated episodes in scenario_1 and reference to scenario_2; 2. IF the sets of episodes are the same THEN remove one scenario;
	5	Titles share the same *Action-Verb* and the direct *Object*;	1. IF the sets of episodes are the same THEN remove one scenario; 2. IF the sets of episodes are not the same THEN rename the Title of one scenario;
Feasibility	1	There are not relationships among scenarios;	Re-write the set of scenarios so that at least a scenario references to another scenarios of the set
	2	Unreachable operations;	Inform the relevant parts of *Episodes or Exceptions*

Table 25 – Recommendations for Analyzing Consistency Properties.

Property	Heuristic	Indicator	Recommendation
Non-interferential	1	Simultaneously enabled operations;	1. Check that all *pre-conditions* or *constraints* associated to the *episode/exception* corresponding to the transition are fulfilled; 2. Notify to the next software development activities;
Boundedness	1	Overflowed resource;	1. Check that the overflowed resource is a critical shared *resource* modified by several scenarios; 2. Notify to the next software development activities;
Reversibility	1	There are no a path from an operation to the initial state;	
Liveness	1	Path to deadlock	1. Check whether there are shared *resources* modified by the scenarios and their relationships; 2. Notify to the next software development activities;
		Never enabled transitions	1. Check that all *pre-conditions, constraints, conditions* or *causes* of the *episode/exception* corresponding to the transition are fulfilled; 2. Notify to the next software development activities;

Following are examples that explain the working of the recommendations in the scenarios of the "Online Broker System". Our analysis approach found a defect that hurts:

- **Vagueness**: The *Episode 4* of the "*Submit Order*" scenario contains a vague term (indicator: "provided"). Therefore, the recommendation of our analysis approach is to "*Remove the vague term*";
- **Simplicity**: The *Episode 1* of the "*Local Supplier bid for order*" scenario contains more than one Action-Verb (Indicator: "receives" and "examines"). Therefore, the recommendation of our analysis approach is to "*Split the episode into multiple episodes*";

Liveness: The Petri-Net corresponding to the "Submit Order" scenario contains a shortest path to deadlock (Indicator: <path from initial state to deadlock>). Therefore, the recommendation of our analysis approach is to "Check whether there are shared resources modified by the scenario and their relationships, and Notify to the next software development activities".

4.7. Final Considerations

We presented a scenario language for describing scenarios using a Restricted-form of Natural Language (RNL). The proposed scenario language enables further transformation of scenario descriptions into executable design models like Petri-Nets, which can be used for more rigorous analysis tasks.

In order to improve the results of analysis of scenario descriptions, we presented heuristics for finding non-explicit relationships among scenarios.

When we use any scenario for analysis, it is required to find and explore the related scenarios. However, it is difficult to ensure that all possible related scenarios are identified. To deal with this problem, we use the heuristics for finding relationships to explore the related scenarios.

We presented an approach for the analysis of scenarios through the use of NLP techniques and Petri-Nets. On the basis of this approach, it is possible to: (1) perform a static analysis to detect defects that hurt properties related to *unambiguity* and *completeness*; (2) perform a dynamic analysis to detect defects that hurt properties related to *consistency*, by executing equivalent Petri-Nets derived from scenarios and their relationships.

Our analysis approach provides *modularity* by first analyzing independent scenarios, then composing related scenarios to one *component*. Also, it supports *traceability*, indicating the defects in Petri-Nets and showing the source of the defects in scenarios (or their relationships).

It is important note that the transformation and integration of Petri-Nets methods do not introduce new defects such as described in Section 4.3.4.

4.7.1. Complexity Analysis

The scenario language does not describe explicit iterations (it does not define "go to"); so that in most of cases, the Petri-Net *PN* derived from a scenario will be an acyclic directed graph. In the Petri-Net derivation method (***Method 1*** - Figure 26), the execution of a sequence of scenario *episodes* or *exceptions* is translated into a sequence of *firing transitions*. In the Petri-Net *PN* all sequence of *firing transitions* are scanned from the *initial places* (initial marking $\boldsymbol{M_0}$) to the *final places* (the set of reachable markings $\boldsymbol{M}$) using the *DFS (Depth-first search)* algorithm. The complexity of DFS is: $O(N+|E|)$ where *'N'* is the *nodes* (|places| + |transitions|) number and *'|E|'* is the *arcs* number. The worst-case order is: $O(N^2)$ where $|E|=N^2$. In scenarios (most case), each node has 2 outgoing edges (if- else for *conditional/optional* episodes), then $|E|=2*N$, and the complexity of DFS be greatly reduced to: $O(|E|) = O(N)$.

Only the integration of Petri-Nets method (***Method 2*** - Figure 27) induces an exponential complexity, because of the number of related Petri-Nets to be synthesized by fusing or substitution places or transitions. However, according to Somé (2010) this number is generally very limited in realistic examples. Thus, the complexity will be greatly reduced. It is O(*N)* when the synthesized Petri-Nets do not interact by non-sequential relationships.

According to Somé (2010), because scenarios describes requirements artifacts, the number of scenarios in projects is typically limited; therefore scaling to much larger projects should not be an issue because of the generally polynomial complexity of Petri-Net transformation and integration methods.

The case studies in Chapter 6 involve projects that specify between 5 and 36 scenarios with different degree of complexity, i.e., every scenario describes between 3 and 12 episodes, between 1 and 5 exceptions, and 1 concurrency construct (#<episode series>#).

5
C&L (Cenários & Léxicos)

In this section we provide a description of ***C&L - Lua*** prototype tool, an integrated environment for supporting the analysis of natural language-based scenarios. It also provides the implementation strategies used to detect violation of properties related to *Unambiguity*, *Completeness* and *Consistency*.

5.1.
C&L

C&L prototype tool was developed at the PUC-Rio Requirements Engineering Group for editing and visualization of natural language-based scenarios and lexicon symbols. The Lua version of C&L (C&L) was developed by Almentero (2009).

Lexicon symbols are described using the Language Extended Lexicon (LEL). LEL is a language designed to help the elicitation and representation of the language used in the application. This model is based on the idea that each application has a specific language. Each symbol in the lexicon is identified by a name or names (synonyms) and two descriptions: Notion (denotation) explains the literal meaning - what the symbol is, Behavioral Response (connotation) describes the effects and consequences when the symbol is used or referenced in the application. Symbols are classified into four types: Subject, Object, Verb and State. Lexicon symbols are referenced within scenario descriptions. Table 26 shows the properties of a LEL symbol.

Table 26 - Symbol definition in lexicon language.

Name	Symbol of LEL
Type	Subject/Object/ Verb/State
Synonymous	Term of LEL/Entry/Symbol
Notion	Word or relevant phrase of the Universe of Discourse. It's described by Name, Type, Notion, Synonymous and Behavioral Response.
Behavioral Response	Its description contains the Type. It has zero or more Synonymous.

In this thesis, lexicon symbols are not considered, but they can be used for further analysis of scenarios against application language represented in the lexicon.

C&L - Lua (Almentero, 2009) is a Web application developed in the Lua programming language (Ierusalimschy, 2013). The Kepler platform (Kepler, 2009) was used to develop C&L because originally Lua was not designed for the development of Web applications. This platform provides a series of modules and tools which facilitates the writing of Lua code for the Web.

The C&L - Lua architecture is based on the Model-View-Controller (MVC) framework. The architecture is vertically divided in layers and horizontally divided in modules. The modules are distributed in the view, controller and model layers, as can be seen in Figure 38. Four main modules were created from the scenarios that describe the situations of the application: User, Project, LEL and Scenario. These modules groups functionalities to manage users (User), projects (Project), lexicon symbols (LEL) and scenarios (Scenario).

The input of the C&L – Lua is composed of projects containing scenarios or lexicons in plain text format. The output is a set of formatted scenarios and lexicons, where the relationships among scenarios or lexicons are represented by hyperlinks. It facilitates the navigation between scenarios and lexicons.

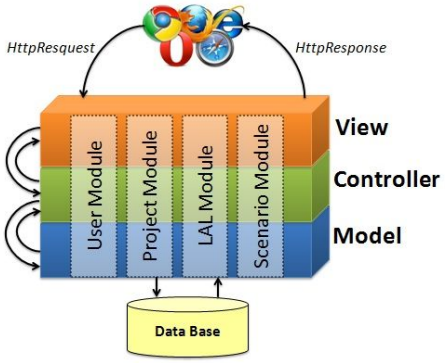


Figure 38 - C&L - Lua Architecture (Sarmiento et al., 2014).

5.2. Extending C&L - Lua

C&L - Lua was extended with the goal to provide an automatic support for the analysis of *unambiguity*, *completeness* and *consistency* qualities of scenarios described using a restricted-form of natural language (Section 4.1, Chapter 4). To reach this goal, we have implemented a set of modules and integrated a set of tools, each one dedicated to a specific analysis purpose of RNL scenarios. In particular, the involved tools are: (1) a NLP tool able to identify *action-verbs*, *subjects* and *objects* involved in scenario sentences; (2) a Petri-Net analysis tool

able to simulate and detect overflows and deadlocks in translated scenarios; and (3) a Network Visualization tool having the aim of visualizing the equivalent Petri-Nets of scenarios. Figure 39 shows the high level architecture of the extended C&L.

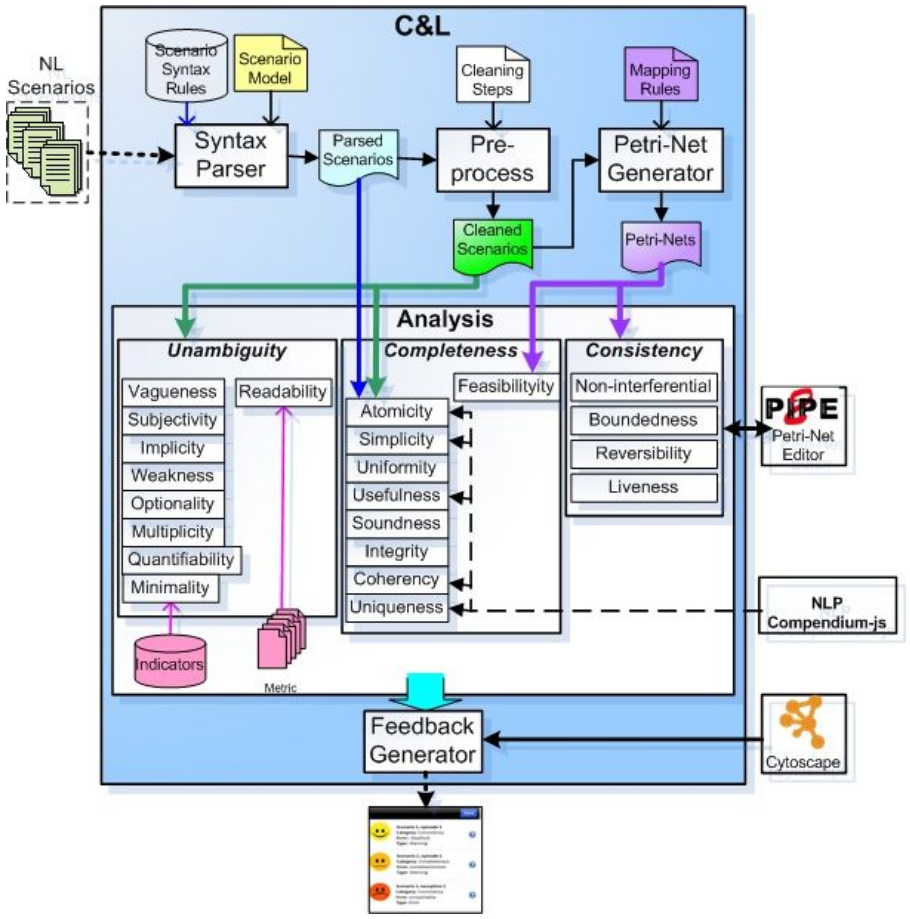


Figure 39 - High Level Architecture of Extended C&L

5.2.1. Tools

During the implementation technologies strictly open source were mainly employed. Below the list of used tools together with short descriptions is presented:

- **NLP Compendium-js**: A Natural-Language-Processing library *in Javascript*, small-enough for the browser, and quick-enough to run on keypress (Compendium-js, 2015). It performs Part-of-Speech tagging (92% on Penn Treebank, 2015), entity recognition, sentiment analysis and more.

- **Cytoscape**: Cytoscape is an open source software platform *in Javascript*, for visualizing complex networks and integrating these with any type of attribute data (Cytoscape, 2015).
- **PIPE2**: An open source tool in Java, for creating and analyzing Petri-Nets (Place/Transition and Generalised Stochastic Petri-Nets). It detects defects that contribute to boundedness, safety and deadlock by analyzing the reachability graph (PIPE2, 2015).

5.2.2. Modules

We extended the C&L - Lua by adding the modules:

- **Syntax Parser**: Chunk textual scenario according to scenario model and syntax. This is necessary to enable the model's transformation.
- **Pre-processing**: Implement the steps for removing irrelevant information from scenario elements.
- **Petri-Net Generator**: Implement the Mapping Rules between scenarios and Petri-Nets.
- **Analysis**: Implement the methods to evaluate Structural (Static analysis) properties of scenarios and Behavioral properties (Dynamic analysis) of equivalent Petri-Nets:
    - **Unambiguity**: Evaluate scenario elements by searching defect indicators and applying metrics. For each property, common defect indicators are stored in *dictionaries* (Chapter 3, Table 7).
    - **Completeness**: Evaluate scenario elements by applying heuristics to search defect indicators (Chapter 3, Table 8, Table 9, Table 10 and Table 11). These heuristics are driven by syntax checks and by cross-referencing the related scenarios. Some properties (atomicity, simplicity, usefulness, conceptually soundness and uniqueness) are evaluated by *Phrase-structure parsing* and using a *NLP tool*.
    - **Consistency**: Evaluate the behavior of a set of related scenarios by running executable equivalent Petri-Nets of scenarios and searching defect indicators (Chapter 3, Table 12). This evaluation is driven by the Reachability Analysis and using a *Petri-Net tool*.

- **Feedback Generator**: Format the output of the analysis module by classifying defects found into *Information*, *Warning* or *Error*; and appointing the defect indicator and the fix recommendation for it. Petri-Net defects are traced into Scenario defects. Additionally, this module uses a *Visualization tool* for enables the visualization of Petri-Nets in a modular way, i.e., Petri-Nets of related scenarios are grouped into separated nets and linked by common places.

5.3. Implementation Details

In this sub section, we discuss the details of each module added to the C&L - Lua. The implementation of each module was driven by scenarios and based on the process proposed by Almentero (2009). Each one of the situations performed by the modules was described using the scenario language presented in this work. The underlined terms (UPPERCASE) are references to other scenarios. Scenarios described in this chapter do not detail exceptional behavior.

The scenarios describing the situations of the remaining modules (User, Project, Lexicon and Scenario) are detailed in Almentero (2009).

5.3.1. Syntax Parser Module

One of the main aspects of verifying whether a plain text represents a scenario description is splitting it into the main scenarios elements (Section 4.1, Chapter 4). This process consist of two steps: (1) Identify the main scenario elements by chunking on common text indicators such as **TITLE**, **GOAL**, **CONTEXT**, **RESOURCE**, **ACTOR**, **EPISODES** and **EXCEPTION**; and (2) Verify that every scenario element contains their relevant components, by chunking on common text indicators such as **GEOGRAPHICAL LOCATION**, **TEMPORAL LOCATION**, **PRE-CONDITION**, **POST-CONDITION**, **CONSTRAINT**, **IF**, **THEN**, **WHILE**, **DO**, **AND**, **OR**, **MUST**, **NOT**, **“[”**, **“]”** and **“#”**.

Each scenario element (Title, Goal, Context, Resource, Actor, Episodes, Exception) is decomposed in its main components and sub-components, i.e., each component is stored in a separate entity. String finding and regular expressions

are used to perform this step. For example, an exception is decomposed in: *Id*, *Cause* (set of conditions) and *Solution*.

5.3.1.1. Construct Scenarios

The main situations to verify the conformance of a plain text to the scenario model presented in Chapter 4 are described as scenarios. Figure 40 describes the steps to identify the scenario elements from a textual scenario.

TITLE: Identify Scenario Elements
GOAL: Produce a parsed scenario.
CONTEXT:
POST-CONDITION: Scenario elements are identified
ACTOR: C&L
RESOURCES: scenario, scenario model
EPISODES
1. The C&L identifies the Title element using the scenario model.
2. The C&L identifies the Goal element using the scenario model.
3. The C&L identifies the Context element using the scenario model.
4. The C&L identifies the Actor element using the scenario model.
5. The C&L identifies the Resource element using the scenario model.
6. The C&L identifies the Episodes element using the scenario model.
7. The C&L identifies the Exception element using the scenario model.
8. The C&L returns the semi-parsed scenario.

Figure 40 – Scenario to Identify the Scenario Elements

The steps to verify that every scenario element contains their main components is described by other scenarios, because each scenario element has particular components described using syntax rules. Figure **41**,

Figure **42**, Figure 43 and Figure 44 describe the situations to verify: Context, Resource, Episodes and Exception elements.

TITLE: Verify Scenario Context
GOAL: Produce a parsed context.
CONTEXT:
PRE-CONDITION: IDENTIFY SCENARIO ELEMENTS
POST-CONDITION: Context components are identified
ACTOR: C&L
RESOURCES: context, scenario syntax
EPISODES
1. The C&L identifies the Pre-conditions in the context using the scenario syntax.
2. The C&L identifies the Post-conditions in the context using the scenario syntax.
3. The C&L identifies the Geographical locations in the context using the scenario syntax.
4. The C&L identifies the Constraints for the Geographical locations using the scenario syntax.
5. The C&L identifies the Temporal locations in the context using the scenario syntax.
6. The C&L identifies the Constraints for the geographical locations using the scenario syntax.
7. The C&L identifies the Constraints for the temporal locations using the scenario syntax.
8. The C&L identifies the Context description using the scenario syntax.
9. The C&L returns the parsed context.

Figure 41 –Scenario to Verify the Main Components of Scenario Context

TITLE: Verify Scenario Resource
GOAL: Produce a parsed resource.
CONTEXT:
PRE-CONDITION: IDENTIFY SCENARIO ELEMENTS
POST-CONDITION: Resource components are identified
ACTOR: C&L
RESOURCES: resource, scenario syntax
EPISODES
1. The C&L identifies the name for each resource using the scenario syntax.
2. The C&L identifies the Constraints for each resource using the scenario syntax.
3. The C&L returns the parsed resource.

Figure 42 –Scenario to Verify the Main Components of Scenario Resource

TITLE: Verify Scenario Episodes
GOAL: Produce a parsed list of episodes.
CONTEXT:
PRE-CONDITION: IDENTIFY SCENARIO ELEMENTS
POST-CONDITION: Episodes components are identified
ACTOR: C&L
RESOURCES: episodes, scenario syntax
EPISODES
1. The C&L identifies the Id for each episode in episodes using the scenario syntax.
2. The C&L identifies the Sentence for each episode in episodes using the scenario syntax.
3. The C&L identifies the Conditions for each episode in episodes using the scenario syntax.
4. The C&L identifies the Constraints for each episode in episodes using the scenario syntax.
5. The C&L identifies the Pre-conditions for each episode in episodes using the scenario syntax.
6. The C&L identifies the Post-conditions for each episode in episodes using the scenario syntax.
7. The C&L returns the parsed episodes.

Figure 43 – Scenario to Verify the Main Components of Scenario Episodes

TITLE: Verify Scenario Exception
GOAL: Produce a parsed list of exceptions.
CONTEXT:
PRE-CONDITION: IDENTIFY SCENARIO ELEMENTS
POST-CONDITION: Exception components are identified
ACTOR: C&L
RESOURCES: exceptions, scenario syntax
EPISODES
1. The C&L identifies the Id for each exception in exceptions using the scenario syntax.
2. The C&L identifies the Solution for each exception in exceptions using the scenario syntax.
3. The C&L identifies the Causes for each exception in exceptions using the scenario syntax.
4. The C&L identifies the Post-conditions for each exception in exceptions using the scenario syntax.
5. The C&L returns the parsed exceptions.

Figure 44 – Scenario to Verify the Main Components of Scenario Exceptions

5.3.1.2. Identify Root Scenario

After scenarios for the Syntax Parser Module were constructed, It is necessary to identify the root scenario of this module. Thus, we first determine the relationship between the scenarios of the module, and from identified relationships we will establish an order between these scenarios. The root scenario will be the scenario that does not require any other scenario of the module (Almentero, 2009).

In this module we identify the relationships shown in

Figure **45**. Based on these relationships we can determine that the scenario "Identify Scenario Elements" must precede all others. With this, we have identified the "Identify Scenario Elements" scenario as the root of this module.

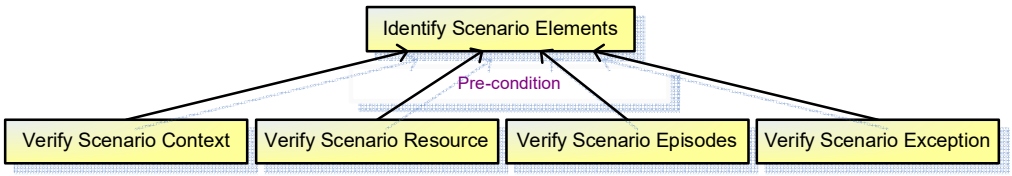


Figure 45 – Relationships among scenarios of Syntax Parser module

5.3.1.3. Construct Integration Scenario

In order to give an overview of the relationship among the several scenarios of the module, we construct an integration scenario. In an integration scenario, an episode corresponds to a sub-scenario**.**

The first step to construct the integration scenario of the module is the identification of the relationships between scenarios and the root scenario, and their order. We can see in

Figure **45** the relationships between the scenarios; the scenarios for verifying scenario elements can be executed in an indistinct or parallel order. Based on the order of the relationships, we create the integration scenario of the module. This scenario can be seen in Figure 46, and "Describe Scenario" scenario is a pre-condition for it (detailed in Almentero, 2009).

TITLE: Parse Scenario
GOAL: Produce a parsed list of exceptions.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is parsed
ACTOR: C&L
RESOURCES: scenario
EPISODES
1. IDENTIFY SCENARIO ELEMENTS
2. #VERIFY SCENARIO CONTEXT
3. VERIFY SCENARIO RESOURCE
4. VERIFY SCENARIO EPISODES
5. VERIFY SCENARIO EXCEPTION#
6. The C&L returns the parsed scenario

Figure 46 – Integration Scenario of Syntax Parser Module

5.3.1.4. Operationalize Scenarios

The scenarios of the Syntax Parser module are implemented by different methods and organized in model layer of the MVC framework.

We utilized *string finding* and *regular expression* matching to perform the sentences described in scenarios' episodes. For instance, two regular expressions are presented below:

- Regular expression used to separate each one of the actors or resources of a scenario:
  - **exp_reg_separ_items = "[%,%;]"**.
  - For example: "actor1, actor2, *actor3*" → {"actor1", "actor2", "actor3"}.

- Regular expression used to identify the ID of each one of the episodes or exceptions of a scenario:
    - **exp_reg_ids** = "[%a*%s*] *%d+[%.%,%:%;%s]%d*[%.%,%:%;%s]%d*[%.%,%;%:]*
    - For example: "1.1. IF Cause1 THEN Exception1" → {id="1.1.", Cause = {"Cause1"}, solution = "Exception1"}.

5.3.2. Pre-processing Module

One of the main aspects for translating scenarios into other models, or for analyzing scenarios using NLP tools, is removing irrelevant information from scenario elements (Chapter 4). This process consists of six steps: (1) Removal of Empty Line; (2) Removal of Capitalization; (3) Removal of Brackets; (4) Removal of URLs; (5) Removal of HTML Markup; and (6) Removal Punctuation**.**

5.3.2.1. Construct Scenarios

Figure 47 presents the scenario that describes the steps to remove the scenario elements of irrelevant information. "Clean Scenario" is the root scenario of this module.

"Clean Scenario" is the root and integration scenario of this module.

TITLE: Clean Scenario
GOAL: Produce a cleaned scenario.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario elements are cleaned
ACTOR: C&L
RESOURCES: scenario
EPISODES
1. The C&L removes the scenario of Empty Line;
2. The C&L removes the scenario of Capitalization;
3. The C&L removes the scenario of Brackets;
4. The C&L removes the scenario of URLs;
5. The C&L removes the scenario of HTML Markup;
6. The C&L removes the scenario of Punctuation;
7. The C&L returns the cleaned scenario.

Figure 47 – Scenario to Clean Scenario of Irrelevant Information

5.3.2.2. Operationalize Scenarios

We utilized *regular expression* matching to perform the actions described in "Clean Scenario" episodes. For instance:

- Regular expression used to remove brackets of a scenario sentence:
    - **remove_parentheses_reg_ex** = '%([^)]*%)'
    - remove_brackets_reg_ex = '%[\[^%]\]*%]'

- remove_braces_reg_ex = '%{[^%}]*%}'
- For example: “The system displays the severity of alarm (high, medium, low).” → “The system displays the severity of alarm.”

- Regular expression used to remove HTML Tags of a scenario sentence:
  - **remove_html_tags_reg_ex** = '(<[^<>]*>)'
  - For example: “The system displays the name of the alarm” → “The system displays the name of the alarm”

- Regular expression used to remove URL of a scenario sentence:
  - **remove_url_reg_exp_http** = '((https?)%:[(%/%/)(%\%\)]+[%w%d%:%#%@%%%/%;%$%(%)%~%_%?%\%+%-%=%!%.%:%,%&]*)'
  - **remove_url_reg_exp_ftp** = '((ftp)%:[(%/%/)(%\%\)]+[%w%d%:%#%@%%%/%;%$%(%)%~%_%?%\%+%-%=%!%.%:%,%&]*)'
  - **remove_url_reg_exp_file** = '((file)%:[(%/%/)(%\%\)]+[%w%d%:%#%@%%%/%;%$%(%)%~%_%?%\%+%-%=%!%.%:%,%&]*)'
  - **remove_email_reg_exp** = '[A-Za-z0-9%.%%%+%-]+@[A-Za-z0-9%.%%%+%-]+%.%w%w%w?%w?'
  - For example: “The system sends an e-mail for the user user@mail.com” → “The system sends an e-mail for the user”

- Regular expression used to remove Punctuation of a scenario sentence:
  - **remove_punctuation_exp_reg** = '[%s+%,%.%:%;%?%!%=%+%-%*%/%#%$%%%&%|%•(%´%s+)(%’%s+)(%`%s+)(%”%s+)(%\'%s+)]'
  - For example: “6. ATM displays ‘New PIN Successful’ message.” → “6. ATM displays New PIN Successful message.”

5.3.3. Petri-Net Generator Module

One of the strategies to simulate a scenario description is mapping it into a Petri-Net (Chapter 4). This process consist of two steps: (1) Translate a scenario

into a Petri-net by using mapping rules; and (2) Integrate the derived Petri-Net with Petri-Nets of related scenarios by using integration rules.

5.3.3.1. Construct Scenarios

Each scenario element (Title, Goal, Context, Resource, Actor, Episodes, Exception) is mapped into Petri-Net nodes (places and transitions) and arcs. Mapping rules are used to perform this task. For example, an exception is mapped into a *transition, input places* and *output places*, which represent the exception solution, causes and post-conditions, accordingly.

Figure 48 presents the scenario that describes the steps to translate a scenario into a Petri-Net.

TITLE: Transform Scenario into Petri-Net
GOAL: Produce an equivalent Petri-Net of a scenario
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is transformed
ACTOR: C&L
RESOURCES: scenario, mapping rules, Petri-Net
EPISODES
1. PARSE SCENARIO
2. CLEAN SCENARIO
3. Map the initial state (title, goal, context, resource, actor) of scenario using mapping rules into sub Petri-Net
4. Map episodes using mapping rules into sub Petri-Net
5. Map concurrency constructs using mapping rules into sub Petri-Net
6. Map exceptions using mapping rules into sub Petri-Net
7. Link the sub Petri-Nets into a whole Petri-Net
8. The C&L returns the Petri-Net of the scenario

Figure 48– Scenario to transform a Scenario into a Petri-Net

A Petri-Net derived from a scenario and the Petri-Nets corresponding to related scenarios are integrated into a whole Integrated Petri-Net. Integration rules are used to perform this task.

Figure 49 presents the scenario that describes the steps to integrate a set of related Petri-Nets.

TITLE: Integrate Petri-Nets
GOAL: Produce an integrated Petri-Net from a root scenario
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is integrated into a whole Petri-Net
ACTOR: C&L
RESOURCES: main scenario, integration rules, Petri-Net
EPISODES
1. PARSE SCENARIO
2. CLEAN SCENARIO
3. TRANSFORM SCENARIO INTO PETRI-NET
4. Identify sequentially related scenarios of main scenario
5. Identify non-sequentially related scenarios of main scenario
6. For each related scenario TRANSFORM SCENARIO INTO PETRI-NET
7. Integrate resulting Petri-Nets into the main Petri-Net
8. The C&L returns the Integrated Petri-Net of the main scenario

Figure 49 – Scenario to integrate a set of related Petri-Nets

5.3.3.2. Identify Root Scenario

In this module we identify the relationships among scenarios described in Figure 48 and Figure 49. Based on these relationships we can determine that both scenarios are root scenarios of this module, because none of them is pre-condition of the other.

5.3.3.3. Construct Integration Scenario

Based on the relationships between scenarios and the root scenarios, we can see in **Figure 49** that episode 3 references the "TRANSFORM SCENARIO INTO PETRI-NET" scenario, the relationships between these scenarios is by *sub-scenario*. Thus, "INTEGRATE PETRI-NETS" scenario is the integration scenario of the module.

5.3.3.4. Operationalize Scenarios

The operationalization of the scenarios to transform a scenario into a Petri-Net and integrate a Petri-Net with related Petri-Nets was presented in Chapter 4: Method 1 - Transform Scenario into Petri-Net and Method 2 - Integrate Petri-Nets.

We utilized *mapping rules*, *fusion/substitution places* and *substitution of transition* operations (Chapter 4: Table 15, Table 16, Table 17, Table 18, Table 19, and Figure 25) to translate a scenario and its relationships into a whole Petri-Net.

5.3.4. Analysis Module

The operationalization of the scenarios to analyze a scenario by detecting defects that hurt Unambiguity, Completeness and Consistency was presented in Chapter 4: Method 3 – Analyze Unambiguity, Method 4 – Analyze Completeness, and Method 5 – Analyze Consistency.

5.3.4.1. Construct Scenarios

Figure 50, Figure 51 and Figure 52 shows the steps to verify the Unambiguity, Completeness and Consistency of a scenario, respectively.

TITLE: Analyze Unambiguity
GOAL: Produce a list of unambiguity defects presents in a scenario.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is analyzed
ACTOR: C&L
RESOURCES: scenario, dictionaries
EPISODES
1. Check Readability index
2. Check Minimality using dictionaries
3. Check Vagueness using dictionaries
4. Check Subjectiveness using dictionaries
5. Check Optionality using dictionaries
6. Check Multiplicity using dictionaries
7. Check Quantifiability using dictionaries
8. Check Weakness using dictionaries
9. Check implicitly using dictionaries
10. The C&L returns the list of defects of the scenario

Figure 50 – Scenario to Analyze Unambiguity

TITLE: Analyze Completeness
GOAL: Produce a list of completeness defects presents in a scenario.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is analyzed
ACTOR: C&L
RESOURCES: scenario, dictionaries, NLP tool, scenario syntax
EPISODES
1. PARSE SCENARIO
2. Check Atomicity using dictionaries and NLP tool
3. Check Uniformity using scenario syntax
4. Check Simplicity using NLP tool
5. Check Usefulness using NLP tool
6. Check Conceptually Soundness
7. Check Coherency
8. Check Uniqueness using NLP tool
9. The C&L returns the list of defects of the scenario

Figure 51 – Scenario to Analyze Completeness

TITLE: Analyze Consistency
GOAL: Produce a list of consistency defects presents in a scenario.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is analyzed
ACTOR: C&L
RESOURCES: scenario, Petri-Net tool
EPISODES
1. PARSE SCENARIO
2. INTEGRATE PETRI-NETS
3. Check Non-deterministic situations using NLP tool
4. Check Deadlock situations using NLP tool
5. Check irreversible situations using NLP tool
6. The C&L returns the list of defects of the scenario

Figure 52 – Scenario to Analyze Consistency

5.3.4.2. Identify Root Scenario

The scenarios of this module do not present relationships, then, we can determine that all scenarios are root scenarios of this module, because none of them is pre-condition of the other.

5.3.4.3. Construct Integration Scenario

Figure 53 presents the integration scenario that describes the steps to analyze a scenario. This scenario references in its episodes the integration scenarios of Syntax Parser, Pre-processing and Petri-Net Generator modules.

TITLE: Analyze Scenario
GOAL: Produce a list of defects presents in a scenario.
CONTEXT:
PRE-CONDITION: DESCRIBE SCENARIO
POST-CONDITION: Scenario is analyzed
ACTOR: C&L
RESOURCES: scenario, dictionaries, NLP tool, Petri-Net tool
EPISODES
1. ANALYZE UNAMBIGUITY using indicators dictionaries
2. ANALYZE COMPLETENESS using NLP tool
3. ANALYZE CONSISTENCY using Petri-Net tool
6. The C&L returns the list of defects of the scenario

Figure 53 – Scenario to Analyze Scenario

5.3.4.4. Operationalize Scenarios

We utilized String finding, Regular expression matching, Phrase-structure parsing, Levenshtein's distance (Levenshtein, 1966), Syntactic similarity heuristic, and Reachability analysis strategies to perform the tasks described in heuristics for finding defects.

5.3.4.4.1. String Finding

The string search operation is used to search a specific string within a scenario sentence. For example, this operation is used to search *ambiguous* indicators in scenario episodes.

- string.find(title, <ambiguous indicator>);

Heuristics for searching defect indicators of *Unambiguity* properties use *String finding* strategy and Coleman-Liau Readability metric.

Figure 54 depicts how Readability index (episode 1) and Weakness indicators (episode 8) are calculated and detected using the Lua language, respectively.

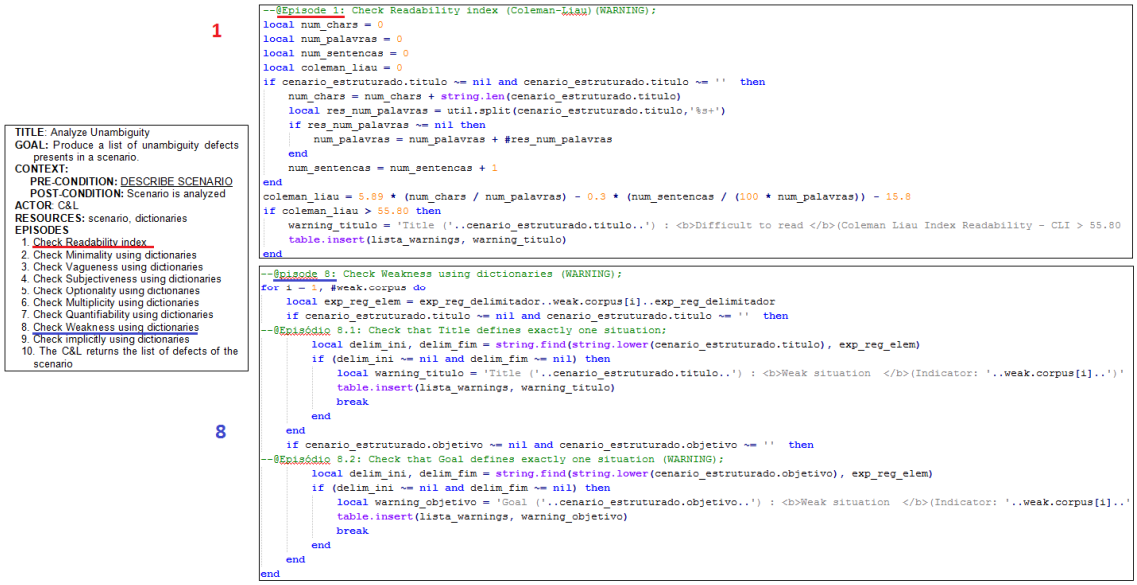


Figure 54 – String Finding Operationalization

5.3.4.4.2. Regular Expression

This is a sequence of characters that forms a search pattern. The search pattern is used for text search in scenario sentences. For example, the following regular expression is used to search extra unnecessary information (text between parentheses) in scenario title:

- search_parentheses_reg_ex = '%([^)]*%)';
- string.find(title, search_parentheses_reg_ex);

5.3.4.4.3. Levenshtein's distance (Levenshtein, 1966)

This strategy is a string metric for measuring the difference between two sequences. The distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. For example, this is used to measure the similarity between two scenarios by comparing their titles or objectives.

- *Distance = levenshtein_distance(<title>, <goal>);*

5.3.4.4.4. Phrase-structure Parsing

This strategy is used to analyze the grammatical structure of sentences, and to identify which words are the ***Subject*** or ***Object*** of a main ***Verb***. This strategy can indicate the forms of nouns and verbs found in a sentence. For example, this

is used to check that scenario title contains an action-verb in infinitive form and an object.

Stanford parser (2015) tool is a most popular program to analyze the grammatical structure of sentences. This tool *chunks* a sentence into POS tags (Klein and Manning, 2003) and presents information about the relations between the *Subject*, *Object* and *Verbs* found in a sentence. In order to improve the accuracy, it is possible to train the parser by providing annotated data.

Figure 55 shows the tags used by NLP tools for tagging words in natural language-based sentences.

CC Coord Conjuncn	and,but,or	PP Personal pronoun	I,you,she
CD Cardinal number	one,two,1,2	PRP$ Possessive pronoun	my,one's
DT Determiner	the,some	RB Adverb	quickly, not
EX Existential there	there	RBR Adverb, comparative	faster
FW Foreign Word	mon dieu	RBS Adverb, superlative	fastest
IN Preposition	of,in,by	RP Particle	up,off
JJ Adjective	big	SYM Symbol	+,%,&
JJR Adj., comparative	bigger	TO 'to'	to
JJS Adj., superlative	biggest	UH Interjection	oh, oops
LS List item marker	1,One	VB verb, base form	eat
MD Modal	can,should	VBD verb, past tense	ate
NN Noun, sing. or mass	dog	VBG verb, gerund	eating
NNP Proper noun, sing.	Edinburgh	VBN verb, past part	eaten
NNPS Proper noun, plural	Smiths	VBP Verb, present	eat
NNS Noun, plural	dogs	VBZ Verb, present	eats
PDT Predeterminer	all, both	WDT Wh-determiner	which,that
POS Possessive ending	's	WP Wh pronoun	who,what
		WP$ Possessive-Wh	whose
		WRB Wh-adverb	how,where

Figure 55 – NLP Tags (Compendium-js, 2015)

Parsing strategy returns a parse tree based on statistical analysis of *POS tags*; however, POS tagging strategies do not performs this task with high precision, such as demonstrated in Table 27. Table 27 shows the POS tagging results returned by Stanford (2015), NLTK (2015) and Compendium-js (2015) tools. The underlined tags are wrong answers pointed out by these tools.

Table 27 – tagging Examples using NLP Tools

NLP tool	***Sentence***	***POS Tags***	***Correct Answer***
Stanford	Process bids	Process/NN bids/NNS	Process/VB bids/NNS
NLTK		Process/NN bids/NNS	
Compendium-js		Process/NN bids/NNS	
Stanford	User downloads the licence file	User/NN downloads/NNS the/DT licence/NN file/NN	User/NN downloads/VBZ the/DT licence/NN file/NN
NLTK		User/NN downloads/NNS the/DT licence/NN file/NN	
Compendium-js		User/NN downloads/NNS the/DT licence/NN file/NN	
Stanford	Administrator types in his user name and password	Administrator/NNP types/NNS in/IN his/PRP$ user/NN name/NN and/CC password/NN	Administrator/NN types/VBZ in/IN his/PRP$ user/NN name/NN and/CC password/NN
NLTK		Administrator/NN types/NNS in/IN his/PRP$ user/NN name/NN and/CC password/NN	
Compendium-js		Administrator/NNP types/VBZ in/IN his/PRP$ user/NN name/NN and/CC password/NN	

In Table 27, it is possible to notice that *Stanford* (2015) and *NLTK* (2015) tools did not identify the main verbs of three sentences. These verbs are tagged as "Nouns": "Process", "Downloads" and "Types". This fact is due to more than one *Part-of-Speech* can be associated with a word. For example in the sentence: *Administrator types in his user name and password*, the word *types* may be interpreted as a noun or verb.

Thus relying only on the parse tree may not provide good accuracy due to the imprecision of POS tagging phase. To improve the accuracy of parsing phase, we provide:

- Adjusting rules based on context-free grammars;
- Adjusting rules based on words that are both "Nouns" and "Verbs";

We noticed that sentences contain words that can be both a "Noun" and a "Verb". In fact, there are many words that can be used to name a person, place or thing and also describe an action. There are many examples of words that can be both nouns and verbs: *"link", "step", "search", "contact", "validate", "approve", "download","store","delete","use","activate","like","form","transfer","view","grant","put","display","broadcast","order","process","bid","prompt","update","access", "account", "release"*. More examples are listed in NounAndAdverb (2015).

Table 28 – Rules to Extract Action-Verbs and Nouns

	Pos Tags	Condition	Rule to adjust Post Tags	Example
Noun	…+ [IN | DT | VB& | RB& | JJ&] + [VB | VBP | VBZ] + …		…+ [IN | DT | VB& | RB& | JJ&] + [NN | NNS] + …	System displays the welcome interface
	…+ [VB | VBP | VBZ]+"OF"+ …		…+ [NN | NNS] + "OF" + …	System displays list of possible criteria
	…+[POS | "'" | "`" | "’"] + [VB | VBP | VBZ] + …		…+[POS | "'" | "`" | "’"] + [NN | NNS] + …	administrator chooses the browse Candidates' list option
	…+[VB | VBP | VBZ]+VB& + …	Second verb is "TO BE" or "TO HAVE"	…+[NN | NNS]+VB& + …	System queries the database for news messages, whose expiry date and time have passed.
	... + [VB | VBP | VBZ] + … + [VB | VBP | VBZ]	Token is a Verb that is also a Noun	... + [VB | VBP | VBZ] + … + [NN | NNS] + "NULL"	User fills all required personal client data forms
Verb	…+ ^[IN | DT | POS | VB& | JJ& | PRP$] + NN& + [IN | DT | VB& | RB& | JJ&] + …	Token is a Noun that is also a Verb	…+ ^[IN | DT | POS | VB& | JJ& | PRP$] + [VB | VBZ] + [IN | DT | VB& | RB& | JJ&] + …	System verifies possibility …
	…+ NN& + ^ (TO + [DT | PDT | PRP$ | NN& | JJ&]) + …	Token is a Noun that is also a Verb	…+ [VB | VBZ] + ^ (TO + [DT | PDT | PRP$ | NN& | JJ&]) + …	User types in the numbers of his PIN and presses the Enter button
	…+ NN& + NN& + (TO + [DT | PDT | PRP$ | NN& | JJ&]) + …	Token is a Noun that is also a Verb	…+ NN& + [VB | VBZ] + (TO + [DT | PDT | PRP$ | NN& | JJ&]) + …	Candidate proceeds to the chosen-majors-view

Table 28 shows the rules to adjust the accuracy of POS tagging phase adding a second phase. In Table 28, + means composition, () is used for grouping, | stands for "OR", [x] denotes the structure to select an option, and "^" denotes

that is not contained within the brackets. “OF” is a terminal word and “…” means that a word is followed by other word.

Parsing strategy was used by Liu et al (2104) and Ciemniewska (2007) to extract action tuples information from use case steps. For improving the accuracy of parsing, they used annotated data for training the analysis on POS tags. However, this task requires an additional manual effort for training phase.

In this thesis, we improved the accuracy of parsing phase by creating simple rules to extract “*Nouns*” and “*Verbs*” based on dictionaries containing words that can be both a “*Noun*” and a “*Verb*”. This strategy adjusts the accuracy of *POS tagging* phase and reduces the manual effort for training.

The tool *Compendium-js* (2015) provides the method “*analyse (sentence)”,* which returns an object containing information like: POS tags and Tokens (Chunked text). Three methods were created for improving the POS tagging and Parsing tree:

- *get_verbs(<sentence>, <analise_sentence>, <verbs_and_nouns>)*: Get action-verbs analysing POS tags, and using adjust rules detailed in Table 28;
- *get_nouns(<sentence>, <analise_sentence>, <verbs_and_nouns>)*: Get nouns analysing POS tags, and using adjust rules detailed in Table 28;
- *get_sentence_components(<sentence>)*: Get parse tree components (Action-verb, Subject and Object) using the action-verbs and nouns returned by previous methods;

Figure 56 depicts the implementation (*JavaScript*) of the method to get the main syntactic components of a textual sentence.

```
/*Title: Get Sentence Components
/*Goal: Get sentence components (Subject, Action-Verb and Objects)
/*Context:
  --Pre-condition: sentence is not empty
/*Actor: C&I
/*Resource: sentence, verbs_and_nouns set, compendium-js
function get_sentence_components(sentence){
   var sentence_components = null;
   var verbs_and_nouns = new Array("finish","link","step", "search","contact","validate","approve","download", "store", "delete", "use",
"signal"); //more in http://www.enchantedlearning.com/wordlist/nounandverb.shtml
   var verbs = new Array();
   var nouns = new Array();
   var subjects = new Array();
   var action_verb = "";
   var verb_time = "";
   var num_sentences = 1;
   var verb = null; //action-verb object
   var objects = new Array();
   sentence = sentence.toLowerCase();
   //@Episode 1: analyse sentence using compendium-js
   var analise_sentence = compendium.analyse(sentence);
   //@Episode 2: get verbs
   verbs = get_verbs(sentence, analise_sentence, verbs_and_nouns);
   //get nouns
   nouns = get_nouns(sentence, analise_sentence, verbs_and_nouns);
   //@Episode 3: get action-verb
   if (verbs != null && verbs.length > 0 ){
            //get verb with the minimal token_index (position in the sentence)
            var position = -1;
            for(var i = 0; i < verbs.length; i++ ){
                     if (i == 0){
                              verb = verbs[i];
                     } else {
                              if(verb.token_index > verbs[i].token_index){
                                       verb = verbs[i];
                              }
                     }
            }
            action_verb = verb.text;
            if(verb.pos == 'VBZ')
                     verb_time = "PRESENT_TENSE"; //VBZ
            else
                     verb_time = "INFINITIVE_FORM"; //VB, VBP

   }
   if (nouns != null && nouns.length > 0 ){
            nouns = concatenate_consecutive_nouns(nouns, verbs);
            for(var i = 0; i < nouns.length; i++ ){
                     if(verb != null){
            //@Episode 4: if there are nouns before 'action-verbs', then add to subjects
                              if (nouns[i].token_index < verb.token_index) {
                                       subjects.push(nouns[i]);
                              }
            //@Episode 5: if there are nouns after 'action-verbs', then add to objects
                              if (nouns[i].token_index > verb.token_index) {
                                       objects.push(nouns[i]);
                              }
                     } else { //no objects
                              //get subjects
                              subjects.push(nouns[i]);
                     }
            }
   }
   //@Episode 6: Get number of sentences
   if(analise_sentence != null && analise_sentence.length > 1){
            num_sentences = analise_sentence.length;
   }
   //@Episode 7: format sentence components
   sentence_components = {
   'action_verb': action_verb, 'verb_time': verb_time, 'subjects': subjects, 'objects': objects, 'verbs': verbs, 'nouns': nouns,
            'num_sentences': num_sentences

   };
   //@Episode 8: return sentence components
   return sentence_components;
}
```

Figure 56 – Get sentence components method (Subject, Action-Verb and Objects).

5.3.4.4.5. Syntactic Similarity Heuristic

This strategy is used to detect the syntactic similarity between two sentences. For example: for each *verb* and *objects* in the title of a scenario, it calculates the similarity with the *verb* and *objects* of another scenario title. We implemented the similarity heuristic by combining related works about modularization of requirements (Al-Otaiby et al., 2005) and similarity in user

stories (Lucassen et al., 2015). We use a similarity measure that produces a value between zero and one, where zero means there is no relationship between the pairs of scenarios under question and one indicates a maximum relationship. The similarity between two scenarios ***i*** and ***j*** is calculated by the following steps:

- **Calculate** *Object_Similarity*(*title(i), title(j)) = m/p* , where *p* is total distinct objects in the two scenarios and *m* is number of matching objects between the scenarios;
- **Find** *Action_Verb(title(i))* and *Action_Verb(title(j))* in the two scenarios;
- **IF** *Object_Similarity*(*title(i), title(j)) > 0* **AND** *Action_Verb(title(i)) = Action_Verb(title(j))* **THEN** Scenario ***i*** and ***j*** are pontentially duplicated;

For example, “International Supplier bid for order” and “Local Supplier bid for order” are potentially duplicated, because they perform the same *Action-Verb* (bid) for the same *Object* (order).

Figure 57 depicts the implementation (*JavaScript*) of the method to measure the syntactic similarity between two textual sentences.

```
*Title: Measure Syntactic Similarity
*Goal: Measure the similarity between two sentences (used to calculate duplicity and coherency)
*Context:
    --Pre-condition: sentence and other_sentence are not empty
*Actor: C&I
*Resource: sentence, other_sentence
unction measure_syntactic_similarity(sentence, other_sentence){
    sentence = sentence.toLowerCase();
    other_sentence = other_sentence.toLowerCase();
    //@Episode 1: Find action-verbs in the sentences
    var nlp_sentence = get_sentence_components(sentence);
    var nlp_other_sentence = get_sentence_components(other_sentence);
    var objects_sentence = new Array();
    var objects_other_sentence = new Array();
    var verbs_sentence = new Array();
    var verbs_other_sentence = new Array();

    if (nlp_sentence != null) {
                objects_sentence = nlp_sentence.objects;
                verbs_sentence = nlp_sentence.verbs;
    } else {
                return false;
    }
    if (nlp_other_sentence != null) {
                objects_other_sentence = nlp_other_sentence.objects;
                verbs_other_sentence = nlp_other_sentence.verbs;
    } else {
                return false;
    }
    //Similarity metric m/p > 0
    var total_objects = 0;
    var total_distinct_objects = 0;
    var total_matching_objects = 0;
    //@Episode 2: Get union of the objects
    var union_objects = new Array();
    if (objects_sentence.length > 0){
                for(var i = 0; i < objects_sentence.length; i++ ){
                            //singulars
                            var singular_noun= objects_sentence[i].stem;
                            union_objects.push(singular_noun);
                            total_objects = total_objects + 1;

                }
    }
    if (objects_other_sentence.length > 0){
                for(var i = 0; i < objects_other_sentence.length; i++ ){
                            //singulars
                            var singular_noun= objects_other_sentence[i].stem;
                            if (contains_verbs_nouns(union_objects, singular_noun) == false){
                                        union_objects.push(singular_noun);
                                        total_objects = total_objects + 1;

                            }
                }
    }
    //@Episode 3: Get number of matching objects between the two sentences ('m')
    if (objects_sentence.length > 0){
                for(var i = 0; i < objects_sentence.length; i++ ){
                            if (objects_other_sentence.length > 0){
                                        for(var j = 0; j < objects_other_sentence.length; j++ ){
                                                    //compare singulars
                                                    if (objects_sentence[i].stem == objects_other_sentence[j].stem){
                                                                total_matching_objects = total_matching_objects + 1;
                                                    }
                                        }
                            }
                }
    }
    //@Episode 4: Get total distinct objects between the two sentences ('p': total objects between )
    total_distinct_objects = total_objects - total_matching_objects;
    //@Episode 5: Check that they have the same action-verb
    var same_action_verb = false;
    if (verbs_sentence.length > 0){
                for(var i = 0; i < verbs_sentence.length; i++ ){
                            if (verbs_other_sentence.length > 0){
                                        for(var j = 0; j < verbs_other_sentence.length; j++ ){
                                                    if (verbs_sentence[i].stem == verbs_other_sentence[j].stem){
                                                                same_action_verb = true;
                                                                break;
                                                    }
                                        }
                            }
                            if (same_action_verb == true){
                                        break;
                            }
                }
    }
                //@Episode 6: IF they have the same action-verb , they are potentially duplicated
    if (same_action_verb && (total_matching_objects/total_distinct_objects) > 0 )
                return true;
    else
                return false;
```

Figure 57 – Syntactic Similarity Implementation.

5.3.4.4.6. Reachability Analysis:

Heuristics for searching defect indicators that hurt *Consistency* properties use *Reachability analysis* strategy implemented in *PIPE2* (2015) tool.

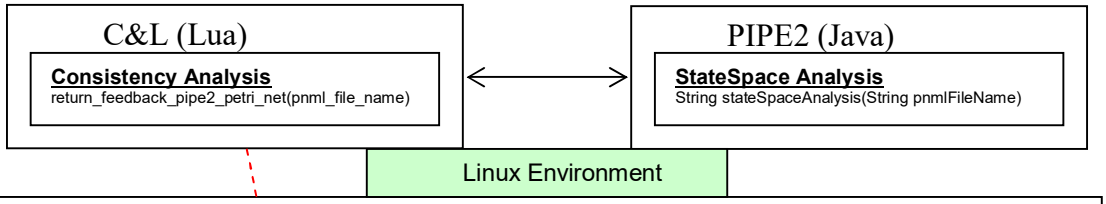
This strategy detects defects in Petri-Nets due to dynamic properties like *Boundedness*, *Liveness* and *Deadlock-free* (Reisig, 1985). In order to detect defects in Petri-Nets and indicate the source of these defects in scenarios and their relationships, we updated the *PIPE2* (2015) tool by adding the "*stateSpaceAnalysis*" method to the module "*StateSpace*".

- *pipe.modules.stateSpace.StateSpace*: This module performs the reachabilty analysis of a Petri-Net in format PNML (Petri-Net Markup language);
- *stateSpaceAnalysis(String pnmlFileName)*: Run state space using as parameter a filename (with pnml format). This method returns a feedback of the analysis, indicating the defects in Petri-Nets and their corresponding scenarios;

Other important method added to the class "*MyTree*" of calculations module is:

- *Boolean[] neverEnabledTransitions()*: This method returns a list containing never enabled transitions, when the Petri-Net is executed.

Figure 58 depicts the implementation (*Lua*) of the method to run a java command line on Linux to run StateSpace analysis method of PIPE2 (2015).




```
[[--
@Title: Generate Consistency Fededback
@Goal: Generate Consistency Fededback of a scenario using its equivalent Petri-Net (pnml_file_name).
@Context:
        - Pré-condition: pnml_file_name is valid
@Atores: C&L, Pipe2.
@Recursos: pnml_file_name
]]--
function return_feedback_pipe2_petri_net(pnml_file_name)
        local retorno_pipe2 = ''
        local xml_retorno_pipe2 = ''
        --@Episode 1: Format the JAVA command to run  pipe2: StateSpace Analysis
        local string cmd_exec_pipe2 = "cd "..retornar_caminho_cel_pipe2().."/target/classes".." && ".."java -classpath .:./lib/jpowergraph-0.2-
common.jar:./lib/jpowergraph-0.2-swing.jar:./lib/powerswing-0.3.jar:./lib/drmaa.jar:./lib/hadoop-0.13.1-dev-core.jar:./lib/jcommon-1.0.10.jar:./lib/jfreechart-
1.0.6.jar:./lib/jfreechart-1.0.6-swt.jar:./lib/tools.jar:.lib/jeval.jar StateSpacePipe ".. "\'"..retornar_caminho_cel_pnml().."/".. pnml_file_name.."\'".."
\'"..retornar_caminho_cel_pnml().."/"..pnml_file_name..".result'"
        --@Episode 2: Run Linux process for StateSpace analysis
        util_exec.exec_silent(cmd_exec_pipe2)
        --@Episode 3: Read the analysis results generated by pipe2
        local file_result = io.open('../visao/pnml/'..pnml_file_name..'.result', "rb") -- r read mode and b binary mode
        if not file_result then
                retorno_pipe2 = 'Error'
        else
                local content = file_result:read "*a" -- *a or *all reads the whole file
                file_result:close()
                retorno_pipe2 = content
        end
        --@Episode 4: Format the feedback in XML format
        xml_retorno_pipe2 = xml_retorno_pipe2.."<pipe2_feedback>"
        if retorno_pipe2 and retorno_pipe2 ~= '' then
                xml_retorno_pipe2 = xml_retorno_pipe2..[[<info>]]
                        xml_retorno_pipe2 = xml_retorno_pipe2..[[<value>]]
                                xml_retorno_pipe2 = xml_retorno_pipe2.."<![CDATA["..retorno_pipe2.."]]>"
                        xml_retorno_pipe2 = xml_retorno_pipe2..[[</value>]]
                xml_retorno_pipe2 = xml_retorno_pipe2..[[</info>]]
        end
        xml_retorno_pipe2 = xml_retorno_pipe2.."</pipe2_feedback>"
        --@Episode 5: Return the feedback
        return retorno_pipe2, xml_retorno_pipe2
    end
```

Figure 58 – Reachability Analysis on PIPE2.

Table 29 shows the implementation strategies for each one of the defect detection heuristics of *Completeness* properties.

5.3.5. Feedback Generator Module

The output of the Analysis module must be formatted and returned to the requirements engineer. Thus, the defects are classified as *Information*, *Warning* or *Error*; and fix recommendations for each defect.

Figure 59 presents the scenario that describes the steps to format the analysis output of a scenario.

5.3.5.1. Construct Scenarios

TITLE: Generate Feedback
GOAL: Produce a formatted list of defects.
CONTEXT:
PRE-CONDITION: ANALYZE SCENARIO
POST-CONDITION:
ACTOR: C&L
RESOURCES: defects
EPISODES
1. The C&L classifies the defects related to Information.
2. The C&L classifies the defects related to Warning.
3. The C&L classifies the defects related to Error.
4. The C&L recommends a fix for each defect.
5. The C&L returns the formatted list of defects.

Figure 59 – Scenario to Generate Feedback

5.3.5.2. Operationalize Scenarios

In Table 29, we detail how each one defect that hurts *Completeness* properties are classified, and how the heuristics to detect them are implemented and classified. The heuristics are classified according to the analysis strategy. The defects are classified as: *Information*, *Warning* or *Error*.

Defects that hurt *Unambiguity* properties are detected by String Finding strategy, and defects that hurt *Consistency* properties are detected by Reachability Analysis of corresponding Petri-Nets.

Table 29 – Intra-scenario Properties Related to Completeness.

Property	Heuristic	Analysis Strategy	Defect Category	Implementation
Atomicity	1	Lexical	Warning	String finding
	2	Lexical	Warning	String finding
	3	Syntactic	Warning	Phrase-structure Parsing
Simplicity	1	Syntactic	Warning	Phrase-structure Parsing
	2	Syntactic	Warning	Phrase-structure Parsing
	3	Lexical	Information	Regular expression matching
	4	Lexical	Warning	String finding
	5	Lexical	Warning	String finding
	6	Lexical	Warning	String finding
Uniformity	1	Lexical	Warning	Regular expression matching, String finding
Usefulness	1	Lexical	Warning	String finding
	2	Syntactic	Warning	Phrase-structure Parsing
	3	Lexical	Warning	String finding
	4	Syntactic	Warning	Phrase-structure Parsing
	5	Lexical	Warning	String finding
	6	Lexical	Warning	String finding
Conceptually Soundness	1	Syntactic	Warning	Syntactic Similarity Heuristic
	2	Semantic	Warning	*Difficult to be measured by an automatic tool;*
	3	Semantic	Warning	
	4	Syntactic	Warning	Phrase-structure Parsing
	5	Lexical	Information	String finding
	6	Lexical	Information	String finding
	7	Lexical	Information	String finding
	8	Syntactic	Warning	Phrase-structure Parsing
	9	Lexical	Information	String finding
Integrity	1	Lexical	Error	String finding
	2	Lexical	Information	String finding
	3	Lexical	Information	String finding
Coherency	1	Semantic	Warning	Difficult to be measured by an automatic tool;
	2	Lexical	Warning	String finding
	3	Lexical	Warning	String finding
Uniqueness	1	Lexical	Warning	Levenshtein's distance
	2	Lexical	Warning	Levenshtein's distance
	3	Lexical	Warning	Levenshtein's distance
	4	Lexical	Warning	Levenshtein's distance
	5	Syntactic	Warning	Syntactic Similarity Heuristic
Feasibility	1	Lexical	Error	Breadth-first search
	2	Lexical	Error	Breadth-first search

5.4. Usage

The extended C&L – Lua comes with simple web-based user interface. In the C&L initial page (Figure 60) the user finds a small text explaining the software and links to external information about the C&L.

To start using the C&L the user must sign up. The user registration is done through a simple form were the user must provide, among other information, its

name, e-mail, login and password. After registering in the application, the user can login informing the necessary data.

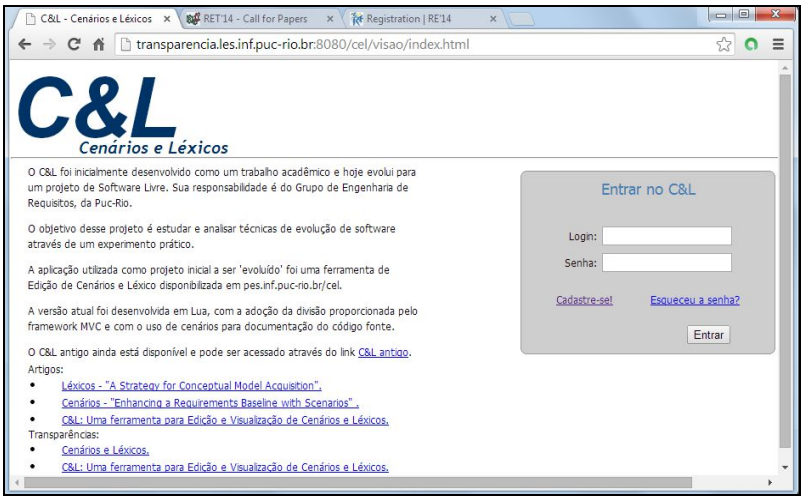


Figure 60 - Initial page of the C&L.

5.4.1. C&L Main Menu

The operation of C&L is described through an integration scenario (Figure 61), which is an artificial scenario created in order to provide an overview of the features of the software. The episodes of this scenario are references to scenarios that describe functionalities provided by the different modules that composes the system. The underlined terms are references to other scenarios (those that appear in uppercase) or to lexicon symbols (those that appear in lowercase). Thus, the term “SELECT PROJECT” in Figure 61 is a link to the scenario that describes how the user selects a project registered in the system. On the other hand, the term “user” that appears several times in the integration scenario, is a link to the description of the lexicon term whose name is “user”. The concept of project is used within the system to represent different domains, where scenarios and lexicon symbols can be grouped.

After the user signing in, the system presents its main interface (Figure 61). In this interface there are many important elements to explore the system functionalities. These elements are: project, lexicon and scenarios menu, and work area. The first element is located at the top right of the interface. It is through this menu that the user selects the project he wants to work with at the moment.

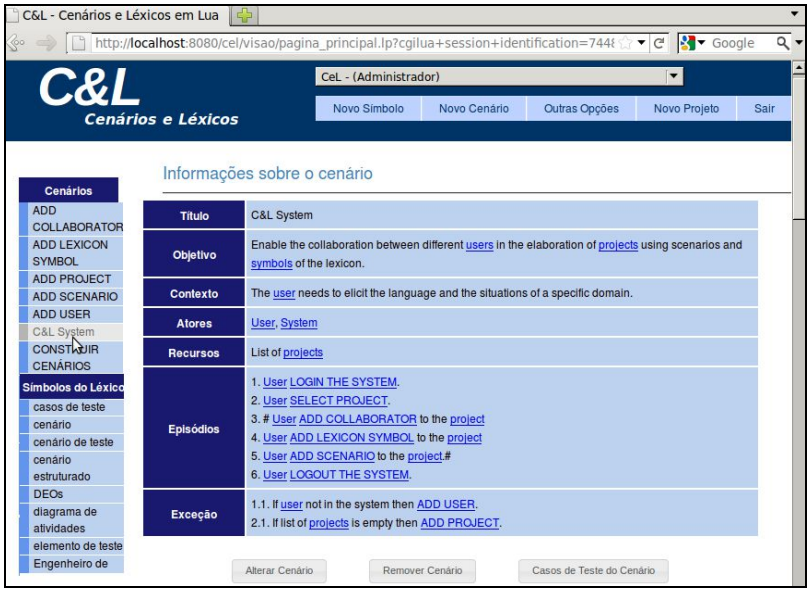


Figure 61 - Integration scenario to use the C&L.

5.4.2. C&L Scenario and Lexicon Functionalities

Three new options appear at the main menu after selecting a project: new lexicon symbol, new scenario and other options. If the user selects the option "new scenario" he is redirected to a form (**Figure 62**) to include the information about the new scenario. The same happens when the "new lexicon symbol" option is selected, but in this case the form allows including information about a lexicon symbol. As the scenarios and lexicon symbols are included in the projects, their title and names are displayed in the lexicons and scenarios menu (left side of the main interface).

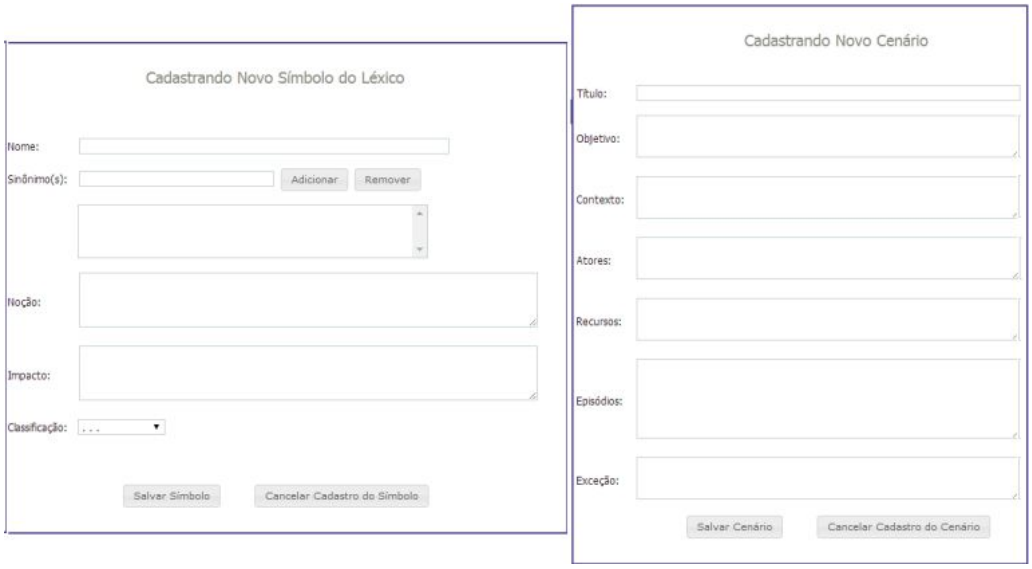


Figure 62 - Add lexicon symbol and add scenario forms.

When the user selects a lexicon symbol or scenario from lexicons and scenarios menu, the C&L application automatically assembles a network of relationships identifying what scenarios and lexicons symbols are referenced in the body of the selected element (Figure 61). The relationships identified are used to create two kinds of trace: backward and forward. These traces allow the navigation between the elements referenced by the one being visualized (forward) and the navigation between the elements that references the one being visualized (backward). The C&L differentiates the links between scenarios (the scenarios are written in uppercase) and lexicon symbols (written in lowercase). The links between scenarios and scenarios, scenarios and lexicon symbols and lexicon symbols and lexicon symbols are created. Those from lexicons to scenarios are not created.

5.4.3. C&L Analysis Functionality

The **scenario analysis** functionality is activated in the project or scenario visualization interfaces (when the user selects a project or a scenario). This functionality generates a feedback containing a list of defects that hurt Unambiguity, Completeness and Consistency properties.

Figure **63** depicts the project visualization interface.

Figure 63 – Visualize Project Form.

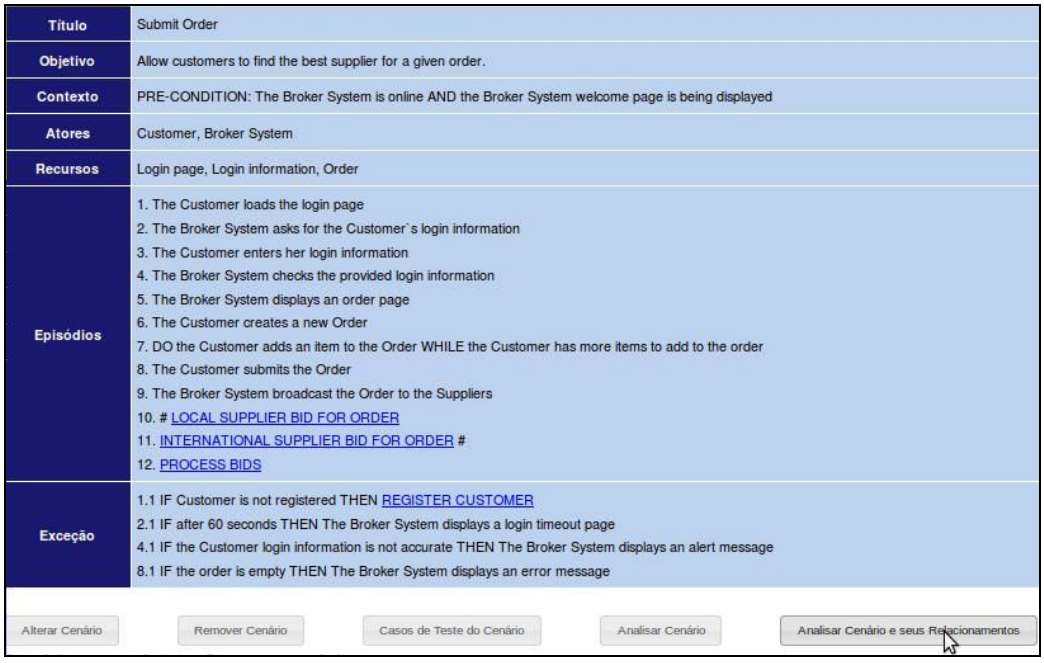


Figure 64 – Visualize Scenario Form.

When the analysis functionality is executed from the project interface, it analyzes each one of the scenarios of the project.

Figure **65** depicts the analysis feedback for the "Online Broker System" project described in Chapter 4.

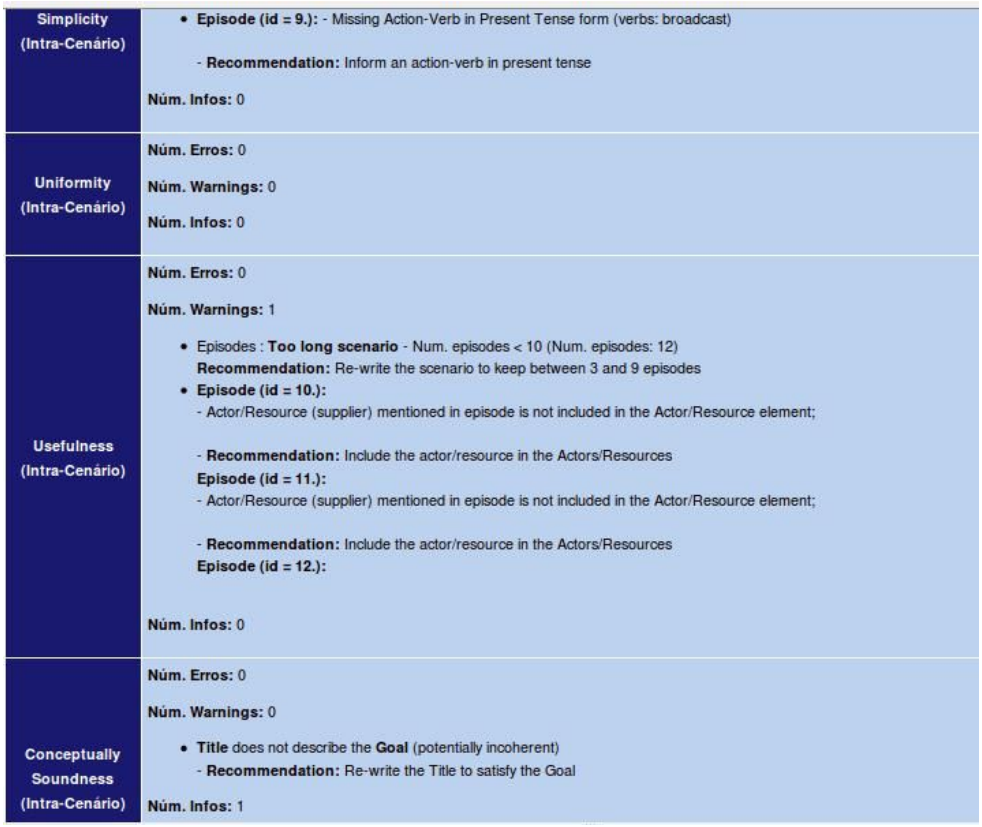


Figure 65 – Project Analysis Feedback Interface (1).

When the analysis functionality is executed from a specific scenario interface, it analyzes the scenario and the related scenarios (sequentially and non-sequentially related). Figure 66 depicts an excerpt of the analysis feedback for the "Submit Order" scenario of the "Online Broker System" project described in Chapter 4.

Figure 66 – Project Analysis Feedback Interface (2).

One important task of the feedback generator modules is to trace the defects reported from the Petri-Net tool (PIPE2, 2015), to the source of these in the scenarios. Figure 66 shows how a path to deadlock in the equivalent Petri-Net of the "Submit Order" scenario, is formatted using scenario elements, i.e. the path to deadlock is presented as a sequence of episodes and exceptions involving related scenarios.

5.4.4. C&L Petri-Net Visualizer Functionality

Other task of the feedback generator module is the visualization of the equivalent Petri-Net. From the analysis feedback form, we can activate the visualization interface. Figure 67 shows an excerpt of the integrated Petri-Net of "Submit Order" scenario. The places and transitions of the related Petri-Nets (related scenarios) are grouped in different modules.

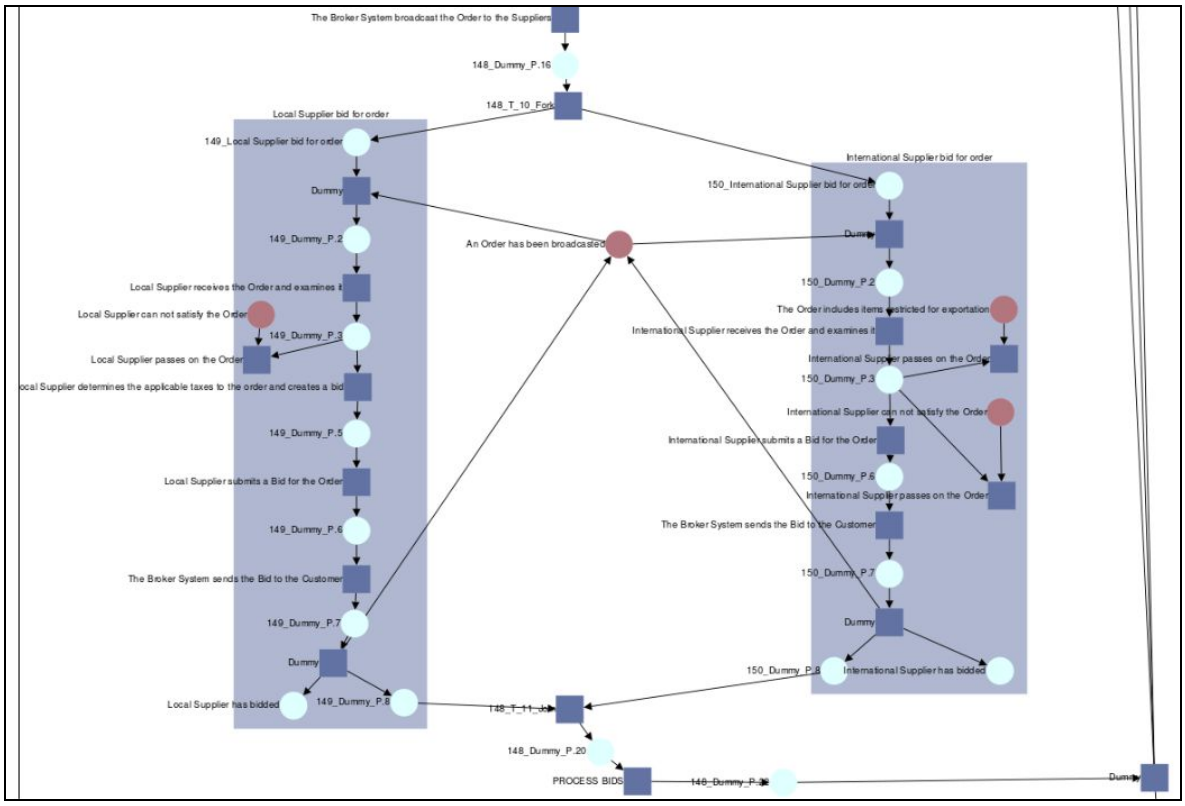


Figure 67 – Petri-Net Visualization Interface.

5.5. Final Considerations

In this chapter we presented details of the implementation of the C&L tool, a tool for supporting the tasks performed by our analysis approach. The objective of the extended C&L is to assist the requirements engineers in their analysis tasks for improving scenario specifications.

Among the limitations of the C&L are the heuristics (for finding defects) that depends on the NLP tool used (Compendium-js, 2015), which is a small version developed in JavaScript. In a further version, it is recommended the use an NLP tool with higher precision like Stanford (2015), openNLP (2015) or GATE (2015).

Other limitation is the method for measuring similarity between scenarios. It uses a syntactical analysis measure (Syntactic Similarity). In future version, we expect to enrich this method by considering synonymous information provided by the WordNet (2015) lexical database.

The implementation was tested on the unit level and on the integration level, using the examples of scenarios presented in case studies. These results will be shown in the next chapter.

6
Case Studies

In order to evaluate the effectiveness of the proposed approach, we apply our analysis approach, implemented in the ***C&L – Lua*** tool, to a set of scenarios from four different SRSs to show which defects are detected.

6.1.
Introduction

This sub section describes the preparation needed to conduct the case studies.

The Goal of our case studies is *to analyze SRSs described as scenarios with the purpose of detecting defects with close to 100% recall and higher precision*. In line with Berry's notion of a dumb RE tool (Berry, 2012).

The case studies address the following research questions:

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

RQ2: Will the *proposed automated analysis approach* detect defects in the SRSs correctly and consistently?

6.1.1.
Hypothesis

The general hypothesis of the case studies is that the *proposed automated analysis approach* should help to identify and show a great deal of defects from a set of scenario specifications, and furthermore take less time than if performed using experts (Requirements Engineers).We aim at evaluating our hypothesis with projects that cover a wide range of software domains, which apply use cases or scenarios for describing requirements, and which are publicly available for other researchers to compare their studies with ours.

6.1.2. Variables

In the case studies, we identified two response variables that will help to corroborate the hypothesis:

- The first variable measures the time needed to identify defects in a set of scenarios (Time Analysis).
- The second variable measures the amount of correct defects identified with our approach (Quality Analysis).

6.1.3. Evaluation Metrics

Regarding the interpretation of the response variables, especially the second one, we chose to apply measures (precision and recall) from Information Retrieval (Olson, 2008). We used the definition of (Alchimowicz, 2011) for the description of the variables used in the *precision* and *recall* definition. Alchimowicz (2011) applied two other metrics: *True positive rate* and *True negative rate*.

Based on (Olson, 2008) we describe Precision and Recall:

- ***Precision*** measures the rate of correct defects identified by the approach (**TP**: *true positives*) in contrast to the amount of incorrect detections (**FP**: *false positives*).
- ***Recall*** measures the rate of correct defects identified by the approach (**TP**) in contrast to the amount of missed defects (**FN**: *false negatives*) from all the defects present in the set of scenarios.

Precision and *Recall* are computed as follows:

$$Precision = TP / TP + FP$$

$$Recall = TP / TP + FN$$

Based on (Alchimowicz, 2011), a defect within a scenario sentence can be classified as following:

- ***True Positive*** *(TP):* A defect is identified by the Requirements engineers and is detected by the approach (Defect occurs).
- ***True Negative*** *(TN):* A defect is not identified by the Requirements engineers and is not detected by the approach (Defect does not occur).

- ***False Positive*** *(FP):* A defect is not identified by the Requirements engineers and is detected by the approach (Defect does not occur).
- ***False Negative*** *(FN):* A defect is identified by the Requirements engineers and is not detected by the approach (Defect occurs).

6.1.4. Case Study Selection

As selection criteria for the four different SRSs, we took into consideration the following parameters:

- *Access to a project in early stages of requirements specifications*: We are interested in projects which apply use cases or scenarios for describing requirements, and that had already a preliminary analysis result, because we need an initial referential solution (baseline) for comparison.
- *Ensuring reasonable diversity of domains*: We are interested in projects that cover a wide range of software domains, i.e., type of project.
- *Ensuring the availability of the project SRS in the community*: We are interested in projects which are publicly available for other researchers to compare their studies.
- *Ensuring reasonable scale*: The number of scenarios must be reasonable.

Several real projects from the literature and their corresponding requirements specifications were analyzed for choosing the ones to be used in our case study that were representative enough for generalizing the findings of this evaluation. It is important to stress that **each selected project did a case study to evaluate** their analysis approach towards improving the quality of the project SRS.

As a result, we selected 4 projects, namely: **Online Broker System** (Somé, 2010), **ATM System** (Cox et al., 2004), **DLibra** (Ciemniewska and Jurkiewicz, 2007), and **Mobile News** (Ciemniewska and Jurkiewicz, 2007). These project's SRSs are described as use cases; so that, in order to evaluate them by our analysis approach, we need to translate them to scenario representations (using the

scenario language proposed in this thesis). This task is carried out before constructing the referential solution (baseline) for each project.

The translation of use cases into scenarios does not introduce new defects, because use case components (pre-condition, post-condition, steps and alternatives) have corresponding components in the scenario language (pre-condition, post-condition, episodes and exceptions), and a sentence is written in a similar basic grammar (a basic sentence is composed of a 3-tuple *subject-verb-object*). **These project's scenarios can be found in the Appendix 1**.

The ***Online Broker System*** consists of use cases, whose goal is to allow customers to find the best supplier (Local and International) for a given order. These use cases were developed by Somé (2010) in order to evaluate an approach to formalize textual use cases using Petri-Net formalism, and detect inconsistencies.

The ***ATM System*** consists of a set of simplified use cases that describe the functionalities (for a supplier) to produce a new cash point for a major bank. There are five use cases described in total: access ATM, withdrawing cash, a customer can check their balance, make a deposit and change their PIN number. These use cases are available in Cox et al. (2004), and the authors introduced defects in order to evaluate a manual inspection technique based on checklists.

The ***DLibra*** and ***Mobile News*** consist of two sets of use cases created as a part of Software Development Studio (SDS) projects. SDS is one of the main components of the Master of Software Engineering program at Poznan University of Technology, Poland. The projects were developed for real customers from the university unit, industry or other organization. Their sources are available in the thesis of Ciemniewska and Jurkiewicz (2007). The case studies of these two projects comprise different software domains: (1) a Web-based Customer Relationship Management (CRM) system for managing the clients of a software for the creation of digital libraries (DLibra CRM), and (2) a news feed system for delivering the latest bulletins to mobile devices. The authors developed use case specifications containing typical defects from industrial projects. Then, they use these use cases to detect common defects, and their results are available in Ciemniewska and Jurkiewicz (2007).

Table 30 summarizes the characteristics of each case study.

Table 30 - Characteristics of the Case Studies

	Broker System (6 scenarios)	ATM System (5 scenarios)	DLibra (15 scenarios)	Mobile News (15 scenarios)	Total (41 scenarios)
Num. of episodes	32	33	80	89	234
Num. of exceptions	9	5	26	5	45
Num. of Pre-condition/ Condition/Cause/Constraint	15	8	33	0	56
Num. of Post-condition	2	9	0	0	11
Total	**58**	**55**	**139**	**94**	**346**

6.1.5. Subjects

In order to create a referential *baseline solution* for each project's case study, we need to construct or validate existing material on the analysis results of each of the four projects, which the respective SRSs were translated to our scenario language.

A referential *baseline solution* lists the defects contained within scenarios or a set of scenarios, which will act as the basis for the evaluation of our analysis approach. A defect in a *baseline solution* is described using the following format: ***<Property> - <Type Defect> : <Detail>***, where "Property" is the quality negatively impacted by the defect, "Type Defect" is the classification of the defect, and "Detail" gives a description of the defect for fixing. **These project's scenarios and baseline solutions can be found in the Appendix 1**.

Thus, to start the evaluation of the project's case studies, we counted on five senior Requirements Engineers as volunteers. These engineers are master and Ph.D Computer Science students at PUC-RIO, 30–40 years old, and they have been working in industry for the last 10 years. Particularly, they have been working with use cases for at least 5 years and with scenarios for at least 2 years.

We applied a questionnaire to collect the experience of these volunteers, which showed that they had similar background; for instance, 100% of them had knowledge about the syntax of Use Cases (Cockburn, 2001) or Scenarios (Leite et al, 2000), and 70% had some knowledge about requirements analysis or inspection. In order to build a shared understanding on their tasks, they received training on our proposed analysis proposal, as well as on the scenario language used in this thesis. These volunteers will construct a referential *baseline solution* for each project's case study. This task is based on the existing preliminary analysis results of each case study.

6.2. Referential Baseline Solution

Whereas the analysis with case studies is comparative in nature, we need to contrast the results obtained with our automated approach (implemented in the *C&L - Lua*) with another one. To keep experimental biases at a minimal level, a valid basis for assessing the analysis results of the case studies must be identified in advance, which act as the ***baseline*** in the evaluation.

The sources and results, with the exception of one, of the SRSs analyzed by related work are available. Somé (2010) only makes available the material analyzed. Cox et al. (2004), and Ciemniewska and Jurkiewicz (2007) make available the material analyzed and their results, i.e., the defects detected by their approaches. Most of the defects contained in the related work preliminary analysis results (set of defects introduced and detected by related work) are mainly related to *Completeness* properties, and a fewer to *Unambiguity* properties.

We used two strategies to establish the ***baselines*** for each selected case study: *Construction* and *Validation*.

We ***construct*** a baseline solution for the "Online Broker System" case study from available scenarios.

We construct the baseline solutions for the "ATM System", "DLibra" and "Mobile News" case studies, by reviewing and ***validating*** the available use cases and preliminary analysis results.

It is not difficult to validate the preliminary analysis results, because the use case language used by related work to write use cases has corresponding components in scenario language, and a sentence is written in a similar basic grammar; such as introduced in previous section.

We detail the process of creation of baseline solutions for each case study in the following Sub-Sections.

6.2.1. Online Broker System (Somé, 2010)

Somé (2010) makes available the set of scenarios formalized using Petri-nets, however, this work does not detect defects in scenarios, and this cannot be directly used as case study. Therefore, we had to carefully define a process that would allow us obtain an objective *baseline* solution.

In order to obtain a baseline solution for the "Online Broke System", the ***subjects*** (volunteers) manually inspected the scenarios, identifying defects across the scenario specifications. The engineers were allocated 1h to perform the analysis of the set of scenarios, each one working separately from the rest. At last, after some discussions, we validated the defects they detected from the scenarios and established a single baseline solution from this case study. The number of defects reported for this case study was larger than we expected.

The obtained *baseline* solution contains defects that hurt *Unambiguity* (Insertion of ambiguous words in sentences), *Simplicity* (action-verb in incorrect tense, missing of the subject or object, sentences containing more than one action-verb), *Usefulness* (actor does not participate in sentences, too short or too long episodes, subjects not described in actor/resource element), and *Uniformity* (incorrect format or missing of the main components in sentences) and *Non-interferential* (simultaneously enabled operations).

6.2.2. ATM system (Cox et al., 2004)

Cox et al. (2004) make available the scenarios analyzed and the defects introduced into the scenarios to manually evaluate an inspection technique. They conducted an experiment with final year undergraduate computer and software engineering students taking a course in Total Quality Management (TQM), at University of New South Wales, National ICT Australia, Sydney, Australia. They introduced several types of defects into the scenarios, including syntactic and semantic defects. Semantics defects are difficult to detect by an automated tool, because this need to understand the meaning of the scenario.

In order to obtain a *baseline solution* for the "***ATM System***", we entrusted the preliminary analysis results of this case study to the selected ***subjects*** (two senior Requirements Engineers).

The requirements engineers received the preliminary analysis results provided by Cox et al. (2004). They manually reviewed and validated the preliminary analysis results of the set of scenarios; they also removed semantic defects detected by Cox et al. (2004). Semantic defects are difficult to detect by an automated tool. The following episode sentence contains a semantic defect, and it was removed:

User selects 'Change PIN'. (Defect: no reference to enter current PIN REQ).

In this example, an automated tool could not detect the missing of a previous episode.

After reviewing and validating defects in preliminary analysis results, the requirements engineers identified some new defects, mainly related to *Unambiguity* properties. The obtained baseline solution contain defects that contribute to *Implicitly* (sentences containing pronouns), *Vagueness* (sentences containing adjectives or adverbs), *Simplicity* (Complex sentences, complex nested conditional sentences, missing action-verb in correct tense or missing object), *Usefulness* (lack of actor or subject in sentences, subjects not described in actor/resource element, too short or long scenarios), *Conceptually Soundness* (missing action-verb in sentences) and *Liveness* (never enabled operations) properties.

6.2.3. DLibra and Mobile News

These projects are detailed in (Ciemniewska and Jurkiewicz, 2007), i.e., the set of scenarios and the common defects introduced in industrial projects are publicized.

In order to obtain baseline solutions for the "***DLibra***" and "***Mobile News***" case studies, we entrusted the preliminary analysis results of this case studies to the selected ***subjects*** (two senior Requirements Engineers). They mapped the ten types of defects detected by the approach proposed by Ciemniewska and Jurkiewicz (2007) into defects that contribute to *Multiplicity*, *Simplicity, Usefulness, Conceptually Soundness, Uniformity* and *Uniqueness* properties of our approach.

The requirements engineers received the preliminary analysis results provided by Ciemniewska and Jurkiewicz (2007). They manually reviewed and validated the preliminary analysis results of the set of scenarios; they also detected defects which do not occur in the referential specification, or are incorrectly classified by Ciemniewska and Jurkiewicz (2007). The following episode sentences contain examples of incorrectly classified defects in the baseline provided by Ciemniewska and Jurkiewicz (2007):

User may sort clients. (Defect: depends on some condition).

User chooses a news group from the 'Today' menu. (Defect: This step is not performed by an actor).

In these examples, these defects are classified incorrectly. In the first sentence, it does not describe a conditional step, however it contains a weak term, then, it is a weak sentence. In the second sentence, it is performed by an actor, then, the defect does not occur.

After reviewing and validating defects in preliminary analysis results, the requirements engineers identified some new defects, mainly related to *Unambiguity* properties. The obtained baseline solution contain defects that contribute to *Multiplicity* (Complex sentences containing more than one sentences), Weakness (sentences containing '*may*' word), *Simplicity* (Complex sentences, complex nested conditional sentences, missing action-verb or missing object), *Usefulness* (lack of actor or subject in sentences, too short or long scenarios, correct step numbering between episodes and exceptions), *Conceptually Soundness* (missing action-verb in sentences) and *Uniformity* (Incomplete exceptions) properties.

6.2.4. Summary of Baselines

Table 31 summarizes the baseline solution for each case study, and lists the number of defects by the quality negatively impacted.

Table 31 – Summary of the Baseline for the Case Studies

	Broker System	ATM System	DLibra	Mobile News	Total
Unambiguity	6	9	35	54	104
Atomicity			1		1
Simplicity	7	7	35	26	75
Uniformity		2		3	5
Usefulness	6	2		3	11
Conceptually Soundness	3				3
Integrity					
Coherency					
Uniqueness	4				4
Non-interferential	2				2
Boundedness					
Reversibility					
Liveness		2			2
Total	**28**	**22**	**71**	**86**	**207**

In Table 31, it is possible to note that most of the defects are related to *Unambiguity* (Insertion of ambiguous words in sentences), *Simplicity* (action-verb in incorrect tense, missing of the subject or object, sentences containing more than

one action-verb), *Usefulness* (actor does not participate in sentences, too short or too long episodes, subjects not described in actor/resource element), and *Uniformity* (incorrect format or missing of the main components in episodes or exceptions).

We put the set of scenarios and baseline solutions of each case study in Appendix 1.

6.3. Evaluation

After baseline solutions for each case study were established by volunteers, we evaluated our scenario analysis approach implemented in the C&L – Lua tool by carrying out the following steps:

- We chose the case studies as the input data for the evaluation (set of scenarios);
- We apply the C&L – Lua to detect defects contained in scenarios of the four case studies;
- We compare the results automatically obtained with the C&L – Lua by looking at the baselines;
- We measure the accuracy of the analysis results using the evaluation metrics (recall and precision);
- We use these measures to answer the response variables that help to corroborate the hypothesis.

6.3.1. Time Analysis

According to Alchimowicz (2011), one of the main characteristics of a tool being developed is its efficiency. Not only developers are interested in this metric, but also for the users, e.g. when requirements engineers have to choose between two tools which give the same results, they would choose the one which is more efficient.

During the evaluation of the case studies, time statistics were gathered to asses the performance of the developed solution to automate the proposed analysis approach.

In the evaluation it was assumed that case study length is the number of sentences (Number of episodes + exceptions + pre-conditions + conditions +

causes + constraints + post-conditions) the set of scenarios is composed of. From the results it can be observed that the analysis can be performed without delays, which would be noticeable for the user. Only processing long case studies (containing from 94 to 139 sentences) takes significant amount of time (as it can be observed at charts in **Figure 68**). Moreover, according to Somé (2010), because scenarios describes requirements artifacts, the number of scenarios and sentences in projects is typically limited.

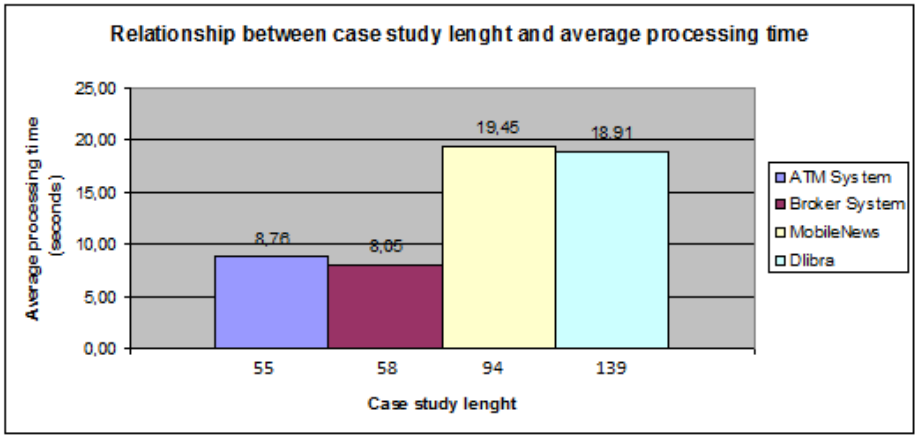


Figure 68 - Relation between case study length and average processing time.

Mobile News (length = 94) case study took more time than DLibra (length = 139) because the length (number of words) of the sentences contained in scenarios of Mobile News is larger than DLibra; so that, POS tagging process of the NLP tool took more time.

6.3.2. Analysis Results

The referential scenario specifications were analyzed by the proposed analysis approach in order to find the defects described for each one of the properties related to Unambiguity, Completeness and Consistency qualities described in Chapter 3.

We apply the ***C&L – Lua*** tool introduced in Chapter 5 to the four scenario sets. The resulting quality analysis shows promising results that indicate high potential for successful further improvements.

Appendix 1 summarizes the data collected from the execution of ***C&L – Lua*** on each of the case studies by comparing the output of our analysis approach against the corresponding baselines. The obtained results (***TP***, ***FP*** and ***FN***) were used to measure the *accuracy* of the ***C&L – Lua*** tool. For each case study, all of

the processed scenarios violate one or more quality properties that the ***C&L – Lua*** can detect.

In the following sub sections are shown the results achieved by the ***C&L – Lua*** on each case study, which indicate the *Recall* and *Precision* of the approach to detect defects that hurt the main quality properties of scenarios.

6.3.2.1. Results of Unambiguity Analysis

As it can be seen from the results in **Table 32**, unambiguity defects were detected with the 100% recall and high precision. Only the precision for ***multiplicity defects*** is under 100% which results from the natural language ambiguities described in Chapter 3.

C&L – Lua achieved a *Precision* above 75%. When this is applied to detect defects related to Multiplicity, the precision is lower (75%). Multiplicity defects are difficult to detect because a sentence can describe several attributes for the same entity, and this fact can be understand by an automated tool as attributes for different entities. For example, in the "Broker System" case study, the following episode sentence is incorrectly recovered (*false positive*) as containing a multiplicity defect (defect indicator: "and"):

"Broker System asks for Customer name, date of birth and address".

In this example, the ***C&L – Lua*** tool understood that name, date of birth and address are attributes of different entities; however, they are attributes of the customer entity.

Table 32 – Analysis of Unambiguity using the C&L – Lua.

Unambiguity Analysis	Broker System		ATM System		DLibra		MobileNews	
	Recall	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**
Vagueness	1	1			1	1	1	1
Subjectiveness								
Optionality								
Weakness					1	1		
Multiplicity	1	0.75	1	1	1	0.86	1	0.97
Implicitly	1	1	1	1	1	1	1	1
Quantifiability					1	1	1	1
Total	**1**	**0.86**	**1**	**1**	**1**	**0.95**	**1**	**0.98**

Overall, the ***C&L – Lua*** produced reasonable results with perfect recall and above 86% precision.

6.3.2.2. Results of Completeness Analysis

From the analysis results of Completeness properties (**Table 33**), one can observe that every analyzed scenario sentence was correctly classified by the ***C&L – Lua*** tool. When it comes to the individual properties *Recall* results, the ***C&L – Lua*** identified the entire set of defects in 3 of the case studies (Broker System, ATM System and DLibra). This result is reflected by a perfect recall score in the **Table 33**. In the "MobileNews" case study, the tool obtained 93% recall.

"MobileNews" results had two *false negatives*, attributed to two *sentences with more than one action-verb* that were not uncovered. Such behavior can be explained by the specificity of English. In English some words can be both verbs and nouns (as described in Chapter 5). This leads to a situation when verb can be confused with noun, for instance in the following sentences:

> "System finds all users matching deletion criteria and deletes found user accounts"
>
> "System receives the confirmation and displays it"

In these examples, the ***C&L – Lua*** understood that "deletes" and "displays" are nouns (NNS).

Regarding the individual properties *Precision* results, the ***C&L – Lua*** obtained acceptable results. In Broker System and DLibra, ***C&L – Lua*** achieved a precision with 100%. In ATM System and MobileNews, ***C&L – Lua*** achieved 75% and 88% precision. This value means that from the complete set of defects identified, only a quarter corresponded to *false positives*.

ATM System and MobileNews results had 4 and 3 *false positives*, caused by the "*Parsing*" strategy used to identify *subjects* and *action-verbs* in sentences (Detailed in Chapter 5). Such behavior is due to the situation when verb can be confused with a noun or adjective, for instance in the following sentences:

> System assigns an expiry date and time to each incoming message
>
> User re-enters new PIN.

In these examples, the "parsing" strategy understood that "time" is a verb (VB), and "re-enters" is an adjective (JJ).

A more detailed analysis revealed that this precision loss was also caused by the "Syntactic Similarity" heuristic used to measure the coherency between the *title* and the *goal* of a scenario (Detailed in Chapter 5). We noticed that some

scenario *titles* and *goals* are written using synonymous terms (i.e., domain entities or actions can be different names or synonymous). Consequently, the detection problem can be attributed to the *syntactic analysis* strategy used by our approach, which is not improved by a semantic strategy. For instance, this situation was encountered in the following sentences:

TITLE: *WITHDRAW CASH*

GOAL: User wants to withdraw money.

In this example, the "cash" and "money" are synonymous, but they are understood as different terms by our approach.

Table 33 – Analysis of Completeness using the C&L – Lua.

Unambiguity Analysis	Broker System		ATM System		DLibra		MobileNews	
	Recall	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**
Atomicity					1	1		
Simplicity	1	1	1	0.88	1	1	0.93	0.93
Uniformity			1	1			1	0.75
Usefulness	1	1	1	1			1	1
Conceptually Soundness	1	1						
Integrity								
Coherency								
Uniqueness	1	1						
Total	**1**	**1**	**1**	**0.73**	**1**	**1**	**0.94**	**0.91**

Overall, the ***C&L – Lua*** produced reasonable results with above 94% recall and 73% precision.

6.3.2.3. Results of Consistency Analysis

Reveal the incorrect behavior (dynamic) of a set of scenarios from initial requirements engineering activities is difficult and hard. This problem can be attributed to the informal nature of scenario descriptions, usually written using natural language. Thus, it is difficult to establish a baseline solution to compare with analysis results obtained using an automated approach.

However, the requirements engineers manually inspected the scenarios of "Broker System" and "ATM System" case studies, and they identified defects like *Non-interferential* and *Liveness* (**Table 31**). In short projects, it can be possible identify *non-deterministic situations* or *never executed sentences*, without executing or simulating the scenarios using rigorous representations.

As it can be seen from the results in **Table 34**, consistency defects were detected with the 100% recall and precision.

Table 34 – Analysis of Consistency using the C&L – Lua.

Unambiguity Analysis	Broker System		ATM System		DLibra		MobileNews	
	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
Non-interferential	1	1						
Boundedness								
Reversibility								
Liveness			1	1				
Total	**1**	**1**	**1**	**1**				

When facing large projects, the number of scenarios could be unmanageable and the requirements engineers have difficulties to manually identify *inconsistencies* from the behavior of scenarios, mainly because the relationships among several scenarios of the project can not be explicit.

With such situations on mind, the ***C&L – Lua*** generates *warning* or *information* messages instead of *error*, to indicate possible behavioral defects. ***C&L – Lua*** can be applied to detect such defects as *non-deterministic situations*, *deadlocks* and *never enabled operations*. That is, it is sufficient to translate related scenarios (identified by our approach, such as demonstrated in Chapter 4) into a consistent Integrated Petri-Net and to perform simulation and reachability analysis over the Petri-Net.

C&L – Lua generates *warning* messages to indicate possible deadlock when two or more scenarios are executed concurrently. Such behavior is due to simultaneously executed scenarios. For instance in the "Broker System", the suppliers are executed concurrently due to common pre-conditions, such behavior can produce a potential deadlock among then. **Figure 69** shows the warning message produced by the ***C&L – Lua*** to indicate a possible path to deadlock when suppliers' scenarios are invoked from the main scenario "Submit Order".

Figure 69 – Consistency Analysis Using Petri-Nets in "Broker System".

"MobileNews" analysis results recovered *Liveness defects*, attributed to 4 exceptions which are never enabled for execution. Such behavior can be detected

by simulating the Petri-Net and detecting never enabled transitions. **Figure 70** shows a never enabled exception from "Delete a User Group" scenario.

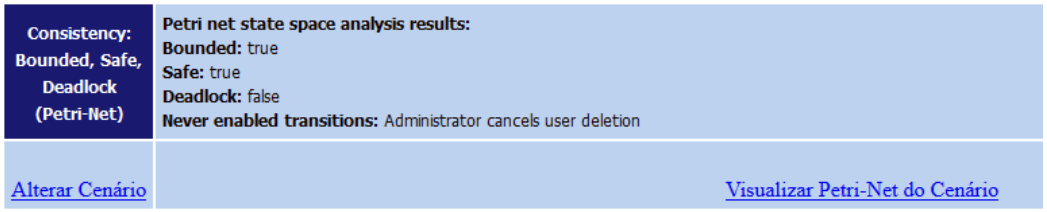


Figure 70 – Consistency Analysis Using Petri-Nets in "Mobile News".

6.3.2.4. Results of Correctness Analysis

Considering that *Correctness* is positively contributed by *Unambiguity*, *Completeness* and *Consistency* qualities; we can aggregate the quality results produced by related properties.

Overall, the automated identification of defects produced reasonable results. Aggregated values of the above accuracy-metrics are as in **Table 35**:

Table 35 – Analysis of Correctness using the C&L – Lua.

Correctness Analysis	Broker System	ATM System	DLibra	Mobile News
Recall	**1**	**1**	**1**	**0.97**
Precision	**0.96**	**0.91**	**0.98**	**0.95**

C&L – Lua detects the total amount of scenarios with defects with above 91% precision and 97% recall.

6.4. Interpretation

Four case studies have been carried out to evaluate the *accuracy* of the proposed analysis approach. These set of scenarios have been evaluated with respect to Information Retrieval metrics. We evaluated the degree of *accuracy* of results produced by the developed tool (C&L-Lua) with respect to reference solutions elaborated by expert Requirements Engineers.

The general hypothesis was verified: "the *proposed automated analysis approach should help to identify and show a great deal of defects from a set of scenario specifications, and furthermore take less time than what it would take to Requirements engineers*". The proposed solution detects defects in scenarios in an acceptable response time, and with close to 100% recall and above 83% precision, inline with Berry's notion (Berry 2012).

The results obtained with the proposed approach implemented in *C&L – Lua* were precise, making only a few mistakes in the detection process. The overall precision was quite high (93% precision), exceeding our expectations. Three out of 4 case studies obtained the maximum recall, which means that, for those defects correctly identified, our approach was able to provide the right advice every time.

The analysis of the evaluation allows to state that the developed methods and heuristics of automatic defects detection are reliable and can significantly improve the quality of the scenario descriptions. Although these set of scenarios describe some abstract systems, they show the typical defects of the industrial requirements specifications, such as demonstrated in (Alchimowicz, 2011).

Finally, case studies show that the developed referential scenario specifications can be used in different ways by both researchers and analysts.

6.4.1. Accuracy of the Petri-Net Generator

We generate one Petri-Net for each scenario; the accuracy of the method to transform a scenario into a Petri-Net is measured by the accuracy of the control flow information of the generated Petri-Net. To be specific, we check (1) whether the *nodes* (places and transitions) and *arcs* are correctly generated and linked in the Petri-Net and (2) whether the *input places* (pre-conditions, conditions, constraints or causes) are correctly associated with the corresponding transitions (episode sentence or exception solution) in the Petri-Net.

According to ***Feasibility*** property of *Completeness* (Chapter 3), it must be possible to perform each operation described in a **scenario** and each internal/external condition must not be violated. *C&L – Lua* evaluates this property by: (1) deriving a Petri-Net for each scenario, and (2) verifying that exist at least a path from the first place (Initial state) to every Petri-Net node. The results obtained shown that all scenarios were correctly translated to Petri-Nets, i.e., they are feasible.

Appendix 1 details the number of Petri-Net input places, transitions and output places generated for each Scenario pre-condition, condition, constraint, cause, episode sentence, exception solution and post-condition, accordingly.

6.4.2. Considerations about Scalability

The case studies considered in this Chapter, involve projects that specify between 5 and 15 scenarios with different degree of complexity, i.e., every scenario describes between 3 and 12 episodes, between 1 and 5 exceptions, and 1 concurrency construct (#<episode series>#). However, in order to evaluate the scalability of our approach it will be necessary to verify the accuracy and response time using larger projects.

In the literature, it is difficult to find publicly available referential specifications to evaluate defect detection approaches. Only, Alchimowicz et al. (2011) make available a referential use case specification containing typical defects in industrial projects. Based on the use cases stored in UCDB (2015), they developed a Referential Specification that has a near-typical profile, that means that its properties is more or less what you can expect to find in real projects. They make available this referential specification to be used for benchmarking tools for use-case analysis.

We used the "Admission System" referential specification (Alchimowicz et al., 2011) to evaluate the scalability of the *C&L – Lua*. However, they do not make available the defects introduced in the use cases to compare with the results obtained by our approach. "Admission System" case study is a system, which enables candidates to apply for the studies through the Internet.

First, I translated the 34 use cases into scenario descriptions; this task is not difficult because use case components (pre-condition, post-condition, steps and extensions) have corresponding components in scenario language (pre-condition, post-condition, episodes and exceptions). **Table 36** shows the characteristics and length of this case study.

Table 36 - Characteristics of the Admission System Case Study

	Admission System (34 scenarios)
Num. of episodes	161
Num. of exceptions	75
Num. of Pre-conditions/ Conditions/Causes	79
Num. of Post-conditions	
Total	**315**

From the evaluation results, we observed that the analysis can be performed without delays. The processing took 1 minute and 35 seconds, and detected defects that hurt *Unambiguity* (Insertion of ambiguous words in sentences),

Simplicity (action-verb in incorrect tense, missing of the subject or object, sentences containing more than one action-verb), *Usefulness* (actor does not participate in sentences, too short or too long episodes), *Uniqueness* (scenarios enabled by the same pre-conditions) and *Liveness* (never enabled exceptions).

This shows that the developed tool for defects detection handled the load, therefore scaling to larger projects may not be an issue because of the generally polynomial complexity of Petri-Net transformation, Petri-Net integration and NLP Syntactic Analysis heuristics implemented in the *C&L - Lua*. However, more studies should be carried out.

Other important strategy to improve the scalability of our analysis approach is the MULTI-STEP BOTTOM-UP consistency analysis approach. This strategy reduces the state explosion of the reachability graph for large and complex Petri-Nets, because it analyses a large system in a compositional way.

6.5. Threats to Validity

In order to reduce the subjectivity of the results obtained in the manual analysis (baseline), we verified that the requirements engineers had similar degrees of background about scenarios or scenarios inspection techniques, by means of a questionnaire. Therefore, more studies should be carried out by requirements engineers with non-homologous profiles.

In order to translate use case descriptions (reported in the related work) into scenario descriptions (proposed in this thesis), manual effort are needed. In this translation process some issues can be introduced, however this is not considered harmful. This task is not difficult because use case components (pre-condition, post-condition, steps and extensions) have corresponding components in scenario language (pre-condition, post-condition, episodes and exceptions), and step/alternative sentences are written in a similar basic grammar (a basic sentence is composed of a 3-tuple *subject-verb-object*).

6.6. Conclusion

In order to corroborate the hypothesis, we used statistical metrics to answer the response variables. Thus, measures of the analysis results are considered reliable.

In order to generalize the results obtained in this evaluation, we used SRSs with different degree of complexity, size, domain, common defects reported in the industry. However more empirical experiment and detailed analysis of proposed heuristics are advisable.

7
Conclusion

Natural language based requirements specification techniques, like the scenario language explored in this work, helps users and developers to improve the quality at early activities of software development process. As such, the analysis of software requirements specifications described as scenario representations improves the quality of the product from the initial stages of software production, contributing to the reduction of failures and the reduction of maintenance costs after the final product was developed.

In this context, it is relevant to develop techniques that enable the automated analysis of these scenarios, so that defects in early requirements artifacts may be identified in way that is more efficient.

Frequently, scenario's representations are normally informal or semi-formal and, in these cases, they cannot be used for further automated analysis. In order to identify defects within scenarios, it is necessary: ***(1)*** to review the scenarios using formal inspection techniques; ***(2)*** to pre-process the scenario descriptions to reduce the ambiguity and analyze them using Natural Language Processing strategies; or ***(3)*** to translate the scenarios from informal to formal representations, like Petri Nets, and simulate the behavior of them to identify consistency defects.

The assessment of quality properties in requirements artifacts has long been investigated in requirements engineering. These are complex concepts, that demands the fulfillment of many others characteristics in order to be achieved.

The main aim of this thesis was to develop an automated approach for detecting defects in scenario specifications to support the analysis of the SRSs. This aim has been achieved by: ***(1)*** modeling and organizing the properties related to scenario's quality; ***(2)*** distinguishing the defect indicators that hurt these quality properties; ***(3)*** proposing operationalizations or heuristics to search these defect indicators; ***(4)*** proposing heuristics for finding non-explicit relationships among scenarios, and improve the analysis results; ***(5)*** investigating Natural Language Processing techniques for automatic detection of syntactic defects; and ***(6)***

investigating rigorous mechanisms to translate scenarios into Petri-Nets and simulate the behavior of a set of related scenarios.

In this thesis, we introduced a novel perception of *Correctness* and its complex relationships with *Unambiguity*, *Completeness* and *Consistency* describing it as a quality that should be satisficed by contributions of related qualities or properties. We also shown how the properties of: ***(1)*** Vagueness, Subjectiveness, Optionality, Multiplicity, Quantifiability, Readabiity, Minimality, Weakness and Implicitly contribute to *Unambiguity*; ***(2)*** Atomicity, Simplicity, Uniformity, Usefulness, Conceptually Soundness, Integrity, Coherency and Uniqueness contribute to *Completeness*; ***(3)*** Non-interferential, Boundedness, Reversibility and Liveness contribute to *Consistency*; and ***(4)*** Unambiguity, Completeness and Consistency contribute to *Correctness*. It means that addressing these properties, we are contributing to requirements *Unambiguity*, *Completeness*, *Consistency*, and consequently to *Correctness*.

As scenarios are written in natural language, we investigated NLP strategies to develop mechanisms of automatic detection of structural or syntactic defects, i.e., these mechanisms verify that scenario sentences are composed of basic attributes like *Subject*, *Action-Verb* and *Object*.

In order to detect defects due to behavioral properties of a set of related scenarios, we proposed an automated strategy to simulate and detect defects from the execution; this strategy is based on the transformation of a set of related scenarios into a whole consistent Petri-Net model. From this transformation, it was possible to: ***(1)*** Define criteria to verify Feasibility, i.e., verify that is possible derive an initial system design from a set of related scenarios; ***(2)*** analyze behavioral properties (like reachability, boundness and liveness) of equivalent Petri-Nets; and ***(3)*** support traceability, detecting the defects in Petri-Net models and indicating the defects within scenarios or in their relationships.

To increase the practical meaning of the proposed methods and heuristics, a prototype solution was developed. The prototype solution was designed and implemented as a set of modules of the ***C&L – Lua*** (Almentero, 2009). This tool assists the requirements engineers during the requirements modeling and analysis phases.

To assess the quality of the results achieved by the C&L – Lua, five case studies were carried out on a set of 75 scenarios with 661 sentences altogether.

The analysis of this evaluation shows that the developed solution is reliable and can improve the requirements quality.

7.1. Comparison with Related Work

Many researches have shown the importance to address the problem of finding defects in early software requirements artifacts written using Natural Language. Some approaches propose *inspection techniques* with quality models to evaluate properties of scenario specifications (Anda and Sjoberg, 2002; Leite et al., 2005; Phalp et al., 2007); usually these approaches are manually applied by requirements inspectors. Other approaches, in order to benefit from automated scenarios analysis, propose the use of *Natural Language Processing* strategies to analyze structural defects in scenario sentences (Ciemniewska and Jurkiewicz, 2007; Liu et al., 2014). Some research focused on developing the *formal semantics* for scenario analysis (Hsia et al., 1994; Cheung et al., 2006). Others are focusing on developing techniques to *translate scenarios* into executable models and detect inconsistencies among scenarios (Lee et al., 1998; Lee et al., 2001; Sinnig et al., 2009; Zhao and Duan, 2009; Somé, 2010).

In (Anda and Sjoberg, 2002; Leite et al., 2005; Phalp et al., 2007) are presented approaches that address the problem of finding defects in scenarios documents with the aid of quality models and inspection techniques. These approaches assess the quality of the documents and to provide hints to potential ambiguities, incompleteness and inconsistencies within the use case descriptions. They are manually performed, and they do not provide insights for further automation. Only Leite et al. (2005) provides some feasible heuristics for finding defects in scenarios.

In (Ciemniewska and Jurkiewicz, 2007; Liu et al., 2014) are proposed approaches to identify syntactic defects in use case documents (semi) automatically with the aid of Natural Language Processing (NLP) techniques. These approaches extract relevant information from scenario sentences and verify that are correctly written. They use *phrase parsing* strategy to identify the *Subject*, *Action-Verb* and *Object* of a use case sentence; and appoint incompleteness or inconsistencies in use case descriptions. However, in order to improve the

accuracy of the phrase parser, manual effort is needed to train the parser by providing annotated data.

In (Lee et al., 1998; Lee et al., 2001; Sinnig et al., 2009; Zhao and Duan, 2009; Somé, 2010) are proposed systematic procedures to convert use case descriptions into Petri-Nets (or their variations) or Labeled Transition Systems - LTS (Sinnig et al., 2009), allowing the analysis of use cases. To facilitate the transformation, use cases are described using a semi-formal syntax. Reachability analysis techniques are used to evaluate *completeness* and *consistency* properties in equivalent Petri-Nets. However, in most of cases, intermediate models are created (Lee et al., 1998; Zhao and Duan, 2009), relationships among use cases are not considered, defects detected in Petri-Nets are not traced to scenarios, and they are not automated. Moreover, only Lee et al. (1998) manages the state explosion issue, a problem of Reachability Analysis.

More details of each one of these approaches are presented in Related Work section of Chapter 2.

In contrast, our approach: (1) defines the properties that contribute to scenarios quality, and describes defect indicators that contribute to these properties; (2) uses a semi-structured natural language to write scenarios; (3) presents heuristics for finding non-explicit relationships among scenarios; (4); takes into consideration the results achieved by **requirement statements** and user story analysis techniques in finding ambiguity indicators; (5) investigates NLP techniques and linguistic characteristics in order to improve the accuracy of parsing strategy; (6) implements automated transformation rules from scenarios into Petri-Nets; (7) preserves the original properties of scenarios when they are translated; (8) identifies potential concurrency defects due to non-sequential relationships; (9) manages the state explosion in Petri-Net analysis; and (10) demonstrates the feasibility of our proposal by implementing the proposed heuristics and methods in a prototype tool. Finally, no additional manual effort is needed for analysis. Table 37 compares our approach with related approaches.

Table 37 - Comparing SRS Analysis Techniques

	Leite et al., 2000	Ciemniewska and Jurkiewicz, 2007	Liu et al., 2014	Lee et al., 1998	Denger et al., 2005	Zhao and Duan, 2009	Sinnig et al., 2009	Somé, 2010	Our approach
Scenario Representation	Scenario	Use Case	Use Case;	Use Case; Action-Condition table;	Use Case;	Use Case;	Use Case; Use Case diagram;	Use Case; Use Case diagram;	RNL Scenario
Syntax for Scenarios	Yes	Partial	Partial	No	Yes	Yes	Yes	Yes	Yes
Analysis Technique	Checklist; Heuristics	Heuristics; NLP;	Checklist; NLP;	Constraints-based Modular Petri-Net;	Checklist; Statechart;	Timed and Controlled Petri-Nets;	LTS;	Reactive Petri-Net;	Checklist; NLP; Place/Transition Petri-Net;
Relationships Among Internal Components	Yes	Yes	Partial	No	Partial	Partial	Yes	No	Yes
Relationships among Scenarios	Yes	Partial	Partial	Yes	Partial	No	Yes	Partial	Yes
Non-explicit Relationships among Scenarios	Manual	Yes	Yes	Yes	No	No	No	No	Yes
Integration of Related Scenarios for Whole Analysis	No	No	No	Yes	Partial	No	Partial	No	Yes
Tool-supported	No	Yes	Yes	No	Partial	Partial	Partial	Yes	Yes
State Explosion Management	---	---	---	Slices	No	No	No	No	Bottom-up approach
Unambiguity	Partial	Yes	Partial	Partial	Partial	Partial	Partial	Partial	Yes
Completeness	Yes	Yes	Partial	Yes	Partial	Partial	Partial	Partial	Yes
Consistency	Yes	Partial	Partial	Yes	Partial	Partial	Partial	Partial	Yes
Correctness	Partial	Partial	No	Yes	Partial	Partial	Partial	Partial	Partial

7.2. Contribution

The proposed approach offers four major contributions:

- **An automated scenarios analysis architecture**: We presented an architecture for scenarios analysis, composed by: 1) *Algorithms to transform scenario descriptions into Petri-Net models*, 2) *Criteria to analyze the unambiguity, completeness, consistency and correctness of scenario descriptions based on analysis of structural and behavioral properties of Petri-Nets*, 3) *Criteria to interpret the results obtained from Petri-Nets analysis* and to allow requirements engineers reduce fault locating time significantly in requirements at early activities of development. Initial results of this approach have been published in (Sarmiento et al. 2015a; Sarmiento et al. 2014e).

- **A restricted-form of natural language for scenarios**: We presented semi-structured linguistic patterns for writing scenario elements, such as

episodes, exceptions, constraints and concurrency; it reduces the ambiguity in natural language scenario descriptions; we also presented formal heuristics for finding non-explicit relationships among scenarios. Initial result has been published in (Sarmiento et al. 2015b).

- **A Quality Model for Scenarios:** The definition of a reusable *Quality Model for Scenarios*, which describes the potential *Defect Indicators* that contribute to properties of the Quality Model from previous related works about requirements statements, user story, use case and scenarios languages. Initial result has been published in (Sarmiento et al. 2015c).
- **Modularity:** A systematic procedure to synthesize a system design from the resulting Petri-Nets of related scenarios, which manages the *State Explosion Issue*. State explosion issue is a serious problem when applying *Petri-Net analysis* to large systems. A contribution of this thesis is a MULTI-STEP BOTTOM-UP analysis approach to manage this problem. Initial result has been published in (Sarmiento et al. 2015a).

Our scenario analysis approach can be applied to a set of scenarios of a project or individual scenarios. When it is applied to a specific scenario, the analysis is carried out on the selected scenario and its related scenarios. So that, the impact among related scenarios are analyzed and defects into the relationships are identified. Therefore, our analysis approach can be used with incremental software development strategies and it also contributes to better understand scenarios evolution and their impacts.

7.3. Limitation

The transformation procedure from scenarios into Petri-Nets works well if a requirements engineer can properly write scenarios using the syntax and semantic rules described in this thesis, i.e. following the linguistic patterns and putting the correct markers (IF THEN, Constraint, and so on) on sentences, It is our assumption that the use of Restricted-form of Natural Language scenarios is well accepted by the most stakeholders in RE process, and it is amenable to automated processing.

The scalability of Petri-Net model and the state explosion of the generated reachability graph are limitations, however these limitations are overcame because

the proposed analysis approach is scalable; the analysis of a large and complex system can be performed in a compositional way, i.e., a MULTI-STEP BOTTOM-UP analysis approach.

The Part-Of-Speech tagger is written in JavaScript programming language, thus the parser memory usage expands roughly with the square of the sentence length. Although, we considered case studies with typical defects and length of industrial scenarios, additional experiments with the requirements taken from the industry could be helpful.

7.4. Future Work

The C&L prototype tool has been used and evolved by the PUC - Rio requirements engineering group. Methods for model transformation and syntactic analysis using NLP are being improved. Their results are positive and therefore its evolution continues.

In the future, we plan investigating other properties related to the main qualities considered; and instantiate the Quality Model used in this thesis to use case language proposed by Cockburn (2001) or its variations.

In a future research, we will explore to enrich our approach by considering semantic analysis. The WordNet (2015) database can be inspected to provide additional information like synonymous of the "*Nouns*" and "*Verbs*" of *POS tagging* phase; and improve the accuracy of the parsing strategy and syntactic similarity heuristic.

Moreover, as a future research direction, we intend to extend the approach to analyze scenarios written in other languages, e.g. Portuguese.

Other future research plan will consider investigating strategies, which automatically traverse the Petri-Net model and its reachability graph to generate test scenarios based on path analysis strategies. This strategy will take into account interactions by "shared resources" or "message passing". Initial results of this approach have been published in (Sarmiento et al. 2015d; Sarmiento et al. 2014e).

References

ALCHIMOWICZ, B.; JURKIEWICZ, J.; NAWROCKI, J.; OCHODEK, M. Towards use-cases benchmark. In: Software engineering techniques. Lecture notes in computer science, Springer-Verlag, v. 4980, Heidelberg, p 20–33, 2011.

ALMENTERO, E.; LEITE, J. C. S. P.; LUCENA, C. Towards Software Modularization from Requirements, In: ACM symposium on Applied Computing, 2014. Proceedings of ACM symposium on Applied Computing, 2014.

ALMENTERO, E. Re-engenharia do software C&L para plataforma Lua-Kepler utilizando princípios de transparência. Master Thesis, PUC-Rio, Brazil, 2009.

AL-OTAIBY, T. N.; ALSHERIF, M.; BOND, W. P. Toward software requirements modularization using hierarchical clustering techniques. In: Annual Southeast Regional Conference, 2005. Proceedings of the 43rd annual southeast regional conference, 2005, p. 223-228.

ANDA, B.; SJØBERG, D. I. Towards an inspection technique for use case models. In: International conference on Software engineering and knowledge engineering, 2002. Proceedings of the 14th international conference on Software engineering and knowledge engineering, 2002, p. 127-134.

ANDA, B.; HANSEN, K.; SAND, G. An investigation of use case quality in a large safety-critical software development project. Information and Software Technology, v. 51, n. 12, p. 1699-1711, 2009.

ANDERSSON, M.; BERGSTRAND, J. Formalizing Use Cases with Message Sequence Charts. 1995. Master's thesis, Lund Inst. of Technology.

ALEXANDER, I. F.; MAIDEN, N. Scenarios, stories, use cases: through the systems development life-cycle. John Wiley & Sons, 2005.

ARORA, C.; SABETZADEH, M.; BRIAND, L.; ZIMMER, F. Automated checking of conformance to requirements templates using natural language processing. IEEE TSE, 2015.

BANSAL, S. Text data cleaning steps python. 2014. Available at: http://www.analyticsvidhya.com/blog/2014/11/text-data-cleaning-steps-python/

BASILI, V.; ROMBACH, H. D. "The TAME Project: Towards Improvement-Oriented Software Environments". IEEE Trans. on Software Engineering, v. 14, n. 6, pp. 758-773, 1988.

BERNSTEIN, L.; YUHAS, C. M. Trustworthy Systems Through Quantitative Software Engineering. Wiley-IEEE Computer Society Press, 2005.

BERRY, D.; GACITUA, R.; SAWYER, P.; TJONG, S. “The Case for Dumb Requirements Engineering Tools”. Requirements Engineering: Foundation for Software Quality, Springer, v. 7195, p. 211–217, 2012.

BOEHM, B. W. "Guidelines for verifying and validating software requirements and design specifications". In: European Conf. Applied Information Technology, 1979. Proceedings of European Conf. Applied Information Technology, 1979, p. 711-719.

BOEHM, B.; BASILI, V. R. Software Defect Reduction Top 10 List. Computer, v. 34, n. 1, p.135-137, 2001.

CABRAL, G.; SAMPAIO, A. Formal specification generation from requirement documents. In: Brazilian Symposium on Formal Methods (SBMF), 2006. Proceedings of Brazilian Symposium on Formal Methods (SBMF), 2006, p. 217–232.

CAMBRIDGE. 2015. Available at: http://dictionary.cambridge.org/grammar/british-grammar/verbs-types

C&L, Scenarios & Lexicons. 2015. Available at: http://pes.inf.puc-rio.br/cel.

CHEUNG, K. S.; CHEUNG, T. Y.; CHOW, K. O. A petri-net-based synthesis methodology for use-case-driven system design. J. Syst. Softw. v. 79, n. 6, p. 772-790, 2006.

CHUNG, L.; NIXON, B.A.; YU, E.; MYLOPOULOS, J. Non-Functional Requirements in Software Engineering. Boston: Kluwer Academic Publishers, 2000.

COHN, M. User Stories Applied: for Agile Software Development. Redwood City: Addison Wesley Longman Publishing Co., Inc., 2004.

COMPENDIUM-JS. 2015. Available at: https://github.com/Ulflander/compendium-js

COX, K.; AURUM, A.; JEFFERY, R. A use case description inspection experiment. In: Australian workshop on software requirements, Sydney, Australia, 2003. Proceedings of Australian workshop on software requirements, 2003.

DAMAS, C.; LAMBEAU, B.; LAMSWEERDE, A. V. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In: ACM SIGSOFT international symposium on Foundations of software engineering, 2006. Proceedings of 14th ACM SIGSOFT international symposium on Foundations of software engineering, ACM, New York, NY, USA, 2006, p. 197-207.

DAMM, W.; HAREL, D. "LSCs: Breathing Life into Message Sequence Charts". Formal Methods in System Design, v. 19, n. 1, p. 45-80, 2001.

DENGER, C.; PAECH, B.; FREIMUT, B. Achieving high quality of use-case-based requirements. Informatik-Forschung und Entwicklung, v. 20, n. 1, p. 11-23, 2005.

DOWNEY, A. B. The Little Book of Semaphores. Green Tea Press, 2005. Available at: http://greenteapress.com/semaphores.

EASTERBROOK, S. “The Difference between verification and validation”, 2010. Available at: http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

ESHUIS, R.; DEHNERT, J. Application and Theory of Petri Nets. LNCS 2679, 2003, p. 295-314.

FEMMER, H.; FERNÁNDEZ, D. M.; JUERGENS, E.; KLOSE, M.; ZIMMER, I.; ZIMMER, J. Rapid requirements checks with requirements smells: two case studies. In: International Workshop on Rapid Continuous Software Engineering, 2014. Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, ACM, 2014, p. 10-19.

GATE. 2015. Available at: https://gate.ac.uk/

GLINZ, M. Improving the quality of requirements with scenarios, In: World Congress for Software Quality(2WCSQ), 2000, Yokohama. Proceedings of the Second World Congress for Software Quality (2WCSQ), 2000, p. 55-60.

GNESI, S.; FABBRINI, F.; FUSANI, M.; TRENTANNI, G. An automatic tool for the analysis of natural language requirements. CSSE Journal, v. 20, n. 1, p. 53-62, 2005.

GRAMMARING. 2015. Available at: http://www.grammaring.com/state-verbs-and-action-verbs

GUTIÉRREZ, J. J.; CLÉMENTINE, N.; ESCALONA, M. J.; MEJÍAS, M.; RAMOS, I. M. Visualization of Use Cases through Automatically Generated Activity Diagrams. In: CZARNECKI, K.; OBER, I.; BRUEL, J. M.; UHL, A.; VÖLTER, M. (eds.) MODELS, 2008, Heidelberg: LNCS, Springer, v. 5301, p. 83–96, 2008.

HAREL, D. StateCharts: a visual formalism for complex systems. SciComput Program, v. 8, n. 3, p. 231-274, 1987.

HEITMEYER, C. Formal methods for specifying, validating, and verifying requirements. J Univ Comput Sci, v. 13, n. 5, 2007, p. 607-618.

HSIA, P.; SAMUEL, J.; GAO, J.; KUNG, D.; TOYOSHIMA, Y.; CHEN, C. Formal Approach to Scenario Analysis. IEEE Software, p. 33-41, 1994.

KELLER, R. Formal Verification of Parallel Programs. Communications of the ACM, v. 19, n. 7, p. 561-572, 1976.

KEPLER PROJECT. 2009. Available at: https://github.com/keplerproject

KLEIN, D.; MANNING, C. D. Accurate unlexicalized parsing. In: Annual Meeting on Association for Computational Linguistics – ACL’03, 2003. Proceedings of the 41st Annual Meeting on Association for Computational Linguistics, 2003, p. 423-430.

IEEE Computer Society. IEEE Recommended Practice for Software Requirements Specications. Technical report, 1998.

IERUSALIMSCHY, R. Programming in Lua. Lua.org, 3 edition, 2013.

ISO/IEC 14977. Extended Backus–Naur Form. 2015. Available at: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

ISO, IEC, and IEEE. ISO/IEC/IEEE 29148:2011. Technical report, ISO IEEE IEC, 2011.

LEE, W.; CHA, S.; KWON, Y. Integration and analysis of use cases using Modular Petri Nets in requirements engineering. IEEE Trans. on Software Engineering, v. 24, n. 12, p. 1115-1130, 1998.

LEE, J.; PAN, J. I.; KUO, J.Y. Verifying scenarios with time petri-nets. Inf. Softw. Technol., v. 43, n. 13, p. 769-781, 2001.

LEITE, J. C. S. P.; HADAD, G; DOORN, J.; KAPLAN, G. A scenario construction process, Requirements Engineering Journal, Springer-Verlag London Limited, v. 5, n. 1, p. 38-61, 2000.

LEITE, J. C. S. P.; DOORN, J. H.; HADAD, G. D.; KAPLAN, G. N. Scenario inspections. Requirements Engineering, v. 10, n. 1, p. 1-21, 2005.

LEITE, J. C. S. P. Livro Vivo: Engenharia de Requisitos. Available at: http://livrodeengenhariaderequisitos.blogspot.com/, 2007.

LEVENSHTEIN, V. I. "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady, v. 10, n. 8, p. 707-710, 1966.

LIU, S.; SUN, J.; LIU, Y.; ZHANG, Y.; WADHWA, B.; DONG, J. S.; WANG, X. Automatic early defects detection in use case documents. In: ACM/IEEE international conference on Automated software engineering, 2014. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, p. 785-790.

LUCASSEN, G.; DALPIAZ, F.; BRINKKEMPER, S.; VAN DER WERF, J. M. E. M. Forging High-Quality User Stories: Towards a Discipline for Agile Requirements. In: IEEE International Requirements Engineering Conference, 2015. Proceedings of the IEEE International Requirements Engineering Conference, 2015, p. 126-135.

MAVIN, A.; WILKINSON, P.; HARWOOD, A.; NOVAK, M. “Easy approach to requirements syntax (EARS)”. In: IEEE International Requirements Engineering Conference (RE’09), 2009. Proceedings of 17th IEEE International Requirements Engineering Conference (RE’09), 2009, p. 317-322.

MURATA, T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, v. 77, n. 4, p. 541-580, 1989.

NLTK. 2015. Available at: http://text-processing.com/demo/tag/

NOUNANDVERB. Words that are also nouns. 2015. Available at: http://www.enchantedlearning.com/wordlist/nounandverb.shtml

OLSON, D. Advanced data mining techniques. Springer Verlag, 2008.

OPENNLP. 2015. Available at: http://opennlp.apache.org/

PHALP, K. T.; VINCENT, J.; COX, K. Assessing the quality of use case descriptions. Software Quality Journal, v. 15, n. 1, p. 69-97, 2007.

PIPE2. Platform Independent Petri net Editor 2. 2015. Available at http://pipe2.sourceforge.net.

POHL, K. The three dimensions of requirements engineering: a framework and its applications. Information Systems Journal, v. 19, n. 3, p. 243-258, 1994.

POHL, K.; RUPP, C. Requirements Engineering Fundamentals, 1st ed. Rocky Nook, 2011.

SARMIENTO, E.; BORGES, M. R. S.; CAMPOS, M. L. M. Applying an event-based approach for detecting requirements interaction, In: International Conference on Enterprise Information Systems (ICEIS 2009), 2009.

SARMIENTO, E.; ALMENTERO, E.; LEITE, J. C. S. P. C&L: Generating Model Based Test Cases From Natural Language Requirements Descriptions In: IEEE International Workshop on Requirements Engineering and Testing - RET'2014, Sweden, 2014.

SARMIENTO, E.; LEITE, J. C. S. P.; RODRIGUEZ, N.; VON STAA, A. An Automated Approach of Test Case Generation for Concurrent Systems from Requirements Descriptions. In: XVI International Conference on Enterprise Information Systems (ICEIS 2014), Portugal, 2014.

SARMIENTO, E.; ALMENTERO, E.; LEITE, J. C. S. P. Analysis of scenarios with Petri-Net models, In: Brazilian Symposium on Software Engineering - SBES, 2015.

SARMIENTO, E.; ALMENTERO, E; LEITE, J. C. S. P.; SOTOMAYOR, G. Mapping Textual Scenario to Analyzable Petri-Nets. In: XVII International Conference on Enterprise Information Systems (ICEIS 2015), Spain, 2015.

SARMIENTO, E.; ALMENTERO, E.; LEITE, J. C. S. P. Using Correctness, Consistency and Completeness Patterns for Automated Scenarios Analysis. In: 5th IEEE Workshop of Requirements Engineering Patterns - RePa, Canada, 2015.

SINHA, A.; SUTTON, S. M.; PARADKAR, A. Text2Test: Automated inspection of natural language use cases. In: International Conference on Software Testing, Verification and Validation (ICST), 2010. Proceedings of Third International Conference on Software Testing, Verification and Validation, 2010, p. 155-164.

SINNIG, D.; CHALIN, P.; KHENDEK, F. LTS semantics for use case models. In: ACM symposium on Applied Computing, 2009. Proceedings of ACM symposium on Applied Computing, 2009, p. 365-370.

SOMÉ, S. S. Formalization of textual use cases based on petri nets. International Journal of Software Engineering and Knowledge Engineering, v. 20, n. 05, p. 695-737, 2010.

SOMMERVILLE, I. Software Engineering. 9 ed. Boston: Addison-Wesley, 2010.

STANFORD. 2015. Available at: http://nlp.stanford.edu

RAGO, A.; MARCOS, C.; DIAZ-PACE, J.A. Identifying duplicate functionality in textual use cases by aligning semantic actions. Software & Systems Modeling, p. 1-25, 2014.

REISIG, W. Petri Nets: An Introduction. Heidelberg: Springer-Verlag, 1985.

ROSCOE, A. W. The Theory and Practice of Concurrency. Prentice Hall, 1998.

ROSS, D. T. Structured Analysis (SA): A Language for Communicating Ideas. IEEE Transactions on Software Engineering, v. 3, n. 1, p. 16-34, 1977.

TJONG, S. F. Avoiding ambiguity in requirements specifications, 2008. PhD thesis, University of Nottingham Malaysia Campus, Faculty of Engineering & Computer Science, Malaysia.

TREEBANK. 2015. Available at: http://www.cis.upenn.edu/~treebank/

UCDB, 2015. Available at: http://www.se.cs.put.poznan.pl/knowledge-base/software-projects-database/use-cases-database-ucdb

UML. Object Management Group, 2015. Available at: http://www.omg.org/spec/UML/

USINGENGLISH. 2015. Available at: http://www.usingenglish.com/glossary/copula-verb.html

VAN LAMSWEERDE, A.; WILLEMET, L. Inferring declarative requirements specifications from operational scenarios. IEEE Transactions on Software Engineering, v. 24, n. 12, p. 1089-1114, 1998.

VAN LAMSWEERDE, A. Goal-oriented requirements engineering: a guided tour. In: Symposium on Requirements Engineering, 2001, Toronto. Proceedings of fifth IEEE International Symposium on Requirements Engineering, 2001, p. 249–262.

WILSON, W. M.; ROSENBERG, L. H.; HYATT, L. E. Automated Analysis of Requirement Specifications. In: International Conference on Software Engineering (ICSE-97), 1997. Proceedings of Nineteenth International Conference on Software Engineering, Boston, 1997.

WORDNET. 2015. Available at: https://wordnet.princeton.edu/

ZHAO, J.; DUAN, Z. Verification of use case with petri nets in requirement analysis. Computational Science and Its Applications-ICCSA, 2009, p. 29-42.

ZOWGHI, D.; GERVASI, V. “On the Interplay Between Consistency, Completeness, and Correctness in Requirements Evolution”. Information and Software Technology, v. 45, p. 993-1009, 2003.

Appendix A1
Referential Specification Used as Baseline

The following scenario descriptions detail the behavior perceived in four systems used in the literature as baseline or referential specification to evaluate the accuracy of defect detection approaches in use cases or scenarios: Online Broker System (Somé, 2010), ATM use cases (Cox et al., 2003), Dlibra CRM (Ciemniewska and Jurkiewicz, 2007) and Mobile News (Ciemniewska and Jurkiewicz, 2007).

The highlighted words or phrases within internal scenario elements (Title, Goal, Context, Resource, Actor, Episodes, Exception), are defect indicators manually detected by Requirements Engineers from the documents, which act as the baseline for the evaluation of our automated analysis approach.

When a defect is detected within scenario element, it is detailed in a new line after the scenario element using the following format: ***(<Property> - <Type Defect> : <Detail>)***, where "Property" is the quality negatively impacted by the defect, "Type Defect" is the classification of the defect, and "Detail" gives a description of the defect for fixing.

Based on Olson (2008), a defect or error can be classified as following:

- *True Positive (TP):* A defect is identified by the experts and is detected by the approach (Defect occurs).
- *True Negative (TN):* A defect is not identified by the experts and is not detected by the approach (Defect does not occur).
- *False Positive (FP):* A defect is not identified by the experts and is detected by the approach (Defect does not occur).
- *False Negative (FN):* A defect is identified by the experts and is not detected by the approach (Defect occurs).

A1.1
The Online Broker System

Table 38 shows the quantitative analysis in scenarios (Pre-conditions, Post-conditions, Episodes, Exceptions) from Online Broker System, and how they are mapped into Petri-Net elements (Input places, Output places, Transitions).

Table 38 - Quantitative Analysis of Online Broke System

ID Scenario	Scenario	Num. Pre-conditions/Conditions/ Causes/Constraints	Num. Post-conditions	Num. Episodes	Num. Exceptions	Num. Input Places - Petri-Net	Num. Transitions - Petri-Net	Num. Output Places - Petri-Net	Num. Dummy Places - Petri-Net
1	Handle Payment	1	0	4	1	1	5	0	7
2	International Supplier bid for order	3	1	3	2	3	5	1	7
3	Local Supplier bid for order	2	1	4	1	2	5	1	7
4	Process Bids	2	0	4	0	2	4	0	6
5	Register Customer	1	0	5	1	1	6	0	8
6	Submit Order	6	0	12	4	6	16	0	18
	Total	**15**	**2**	**32**	**9**	**15**	**41**	**2**	**53**

Table 39 shows the Unambiguity analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Online Broker System. Let ***TP*** be the number of defects detected correctly by out analysis approach; ***FP*** be the number of defects detected incorrectly (defect does not occur); ***FN*** be the number of defects that are not detected (defect occurs).

Table 39 – Unambiguity Analysis of Online Broke System

Scenario	Vague			Subjective			Optional			Weak			Multiple			Implicit			Quantifiable		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1																1					
2													1								
3													2								
4																					
5														1							
6	1															1					
Total	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**1**	**0**	**2**	**0**	**0**	**0**	**0**	**0**

Table 40 shows the Completeness analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Online Broker System.

Table 40 - Completeness Analysis of Online Broke System

ID Scenario	Atomicity			Simplicity			Uniformity			Usefulness			Conceptually Soundness			Integrity			Coherency			Uniqueness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1				1																				
2				1						1			1									2		
3				2						1			1									2		
4				1						1														
5				1																				
6				1						3			1											
Total	**0**	**0**	**0**	**7**	**0**	**0**	**0**	**0**	**0**	**6**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**4**	**0**	**0**

Table 41 shows the *Consistency* analysis (qualitative) in scenarios and related scenarios from Online Broker System.

Table 41 - Consistency Analysis of Online Broke System

ID Scenario	Non-interferential			Boundedness			Reversibility			Liveness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1												
2	1											
3	1											
4												
5												
6												
Total	2	0	0	0	0	0	0	0	0	0	0	0

TITLE: ***Handle Payment***
GOAL: Handle Payment
CONTEXT: Handle payment for a Bid
PRE-CONDITION:
POST-CONDITION:
ACTOR: Customer, Broker System, Payment System
RESOURCES: Credit card information
EPISODES
1. The Broker System asks the Customer for Credit Card information
2. The Customer provides her Credit Card information
(Implicit - TP: ambiguous indicator)
3. The Broker System asks a Payment System to process the Customer's Payment.
(Simplicity - TP: Contains more than one Action-Verb)
4. The Broker System displays an acknowledgement message to the Customer
EXCEPTIONS
1.1 **IF** The Customer Payment is denied **THEN** The Broker System displays a payment denied page

TITLE: ***International Supplier bid for order***
(Soundness - TP: Title does not describe the Goal)
(Non-interferential - TP: Simultaneous enabled with Local Supplier)
(Uniqueness - TP: Local Supplier bid for order and International Supplier bid for order are potentially duplicated)
GOAL: Submit a bid
CONTEXT: Create a Bid for an Order
PRE-CONDITION: An Order has been broadcasted
(Uniqueness - TP: Context Pre-condition coincidence with Related scenario's Local Supplier)
POST-CONDITION: International Supplier has bidden
ACTOR: International Supplier, Broker System
RESOURCES: Order, Bid
EPISODES
1. International Supplier receives the Order and examines it
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
2. International Supplier submits a Bid for the Order
3. The Broker System sends the Bid to the Customer
(Usefulness - TP: Actor/Resource mentioned in episode is not included in the Actor/Resource element)
EXCEPTIONS
1.1 **IF** The Order includes items restricted for exportation **THEN** International Supplier passes on the Order
1.2 **IF** International Supplier can not satisfy the Order **THEN** International Supplier passes on the Order

TITLE: ***Local Supplier bid for order***

(Soundness - TP: Title does not describe the Goal)
(Non-interferential - TP: Simultaneous enabled with International Supplier)
(Uniqueness - TP: Local Supplier bid for order and International Supplier bid for order are potentially duplicated)
GOAL: Submit a bid
CONTEXT: Create a Bid for an Order
PRE-CONDITION: An Order has been broadcasted
(Uniqueness - TP: Context Pre-condition coincidence with Related scenario's International Supplier)
POST-CONDITION: Local Supplier has bidden
ACTOR: Local Supplier, Broker System, Customer
RESOURCES: Order, Bid
EPISODES
1. Local Supplier receives the Order and examines it
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
2. Local Supplier determines the applicable taxes to the order and creates a bid.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
3. Local Supplier submits a Bid for the Order
4. The Broker System sends the Bid to the Customer.
(Usefulness - TP: Actor/Resource mentioned in episode is not included in the Actor/Resource element)

EXCEPTIONS
1.1 **IF** Local Supplier can not satisfy the Order **THEN** Local Supplier passes on the Order

TITLE: ***Process Bids***
GOAL: Process a bid
CONTEXT: Process a Bid for an Order
PRE-CONDITION: Local Supplier has bidden **OR** International Supplier has bidden
ACTOR: Customer, Broker System
(Usefulness - TP: never participates in episodes)
RESOURCES: Order, Bid
EPISODES
1. Customer examines the bid
2. Customer signals the system to proceed with bid
3. HANDLE PAYMENT
4. System put an order with the selected bidder
(Simplicity - TP: Missing Action-Verb in Present Tense form)

TITLE: ***Register Customer***
GOAL: Register Customer
CONTEXT: login page loaded
PRE-CONDITION:
POST-CONDITION:
ACTOR: Customer, Broker System
RESOURCES: registration operation, name, date of birth, address, login information
EPISODES
1. Customer selects registration operation
2. Broker System asks for Customer name, date of birth and address
(Multiplicity - FP: ambiguous indicator)
3. Customer enters registration information
4. Broker System validates Customer information
5. Broker System generate login information for Customer
(Simplicity - TP: Missing Action-Verb in Present Tense form)

EXCEPTIONS
4.1. IF Customer registration information is not valid THEN Broker System displays registration failure page

TITLE: **Submit Order**

(Soundness: Title does not describe the Goal)
GOAL: Allow customers to find the best supplier for a given order.
CONTEXT:
PRE-CONDITION: The Broker System is online **AND** the Broker System welcome page is being displayed
ACTOR: Customer, Broker System
RESOURCES: Login page, Login information, Order
EPISODES
(*Usefulness - TP: Too long scenario - Num. episodes > 10)*
1. The Customer loads the login page
2. The Broker System asks for the Customer's login information
3. The Customer enters her login information
(Implicit - TP: ambiguous indicator)
4. The Broker System checks the provided login information
(Vagueness - TP: ambiguous indicator)
5. The Broker System displays an order page
6. The Customer creates a new Order
7. **DO** the Customer adds an item to the Order **WHILE** the Customer has more items to add to the order
8. The Customer submits the Order
9. The Broker System broadcast the Order to the Suppliers
(Simplicity - TP: Missing Action-Verb in Present Tense form)
10. # LOCAL SUPPLIER BID FOR ORDER.
(Usefulness - TP: Actor/Resource mentioned in episode is not included in the Actor/Resource element)
11. INTERNATIONAL SUPPLIER BID FOR ORDER #
(Usefulness - TP: Actor/Resource mentioned in episode is not included in the Actor/Resource element)
12. PROCESS BIDS
EXCEPTIONS
1.1 **IF** Customer is not registered **THEN** REGISTER CUSTOMER
2.1 **IF** after 60 seconds **THEN** The Broker System displays a login timeout page.
4.1 **IF** the Customer login information is not accurate **THEN** The Broker System displays an alert message
8.1 **IF** the order is empty **THEN** The Broker System displays an error message

A1.2
The ATM System

Table 42 shows the quantitative analysis in scenarios (Pre-conditions, Post-conditions, Episodes, Exceptions) from ATM System , and how they are mapped in Petri-Net elements (Input places, Output places, Transitions).

Table 42 - Quantitative Analysis of ATM System

ID Scenario	Scenario	Num. Pre-conditions/Conditions/ Causes/Constraints	Num. Post-conditions	Num. Episodes	Num. Exceptions	Num. Input Places - Petri-Net	Num. Transitions - Petri-Net	Num. Output Places - Petri-Net	Num. Dummy Places - Petri-Net
1	ACCESS ATM	1	3	6	1	1	7	3	9
2	CHANGE PIN	2	2	7	1	2	8	2	10
3	CHECK BALANCE	2	2	5	1	2	6	2	8
4	MAKE DEPOSIT	2	1	4	1	2	5	1	7
5	WITHDRAW CASH	1	1	11	1	1	12	1	14
	Total	**8**	**9**	**33**	**5**	**8**	**38**	**9**	**48**

Table 43 shows the Unambiguity analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from ATM System. Let ***TP*** be the number of defects

detected correctly by out analysis approach; ***FP*** be the number of defects detected incorrectly (defect does not occur); ***FN*** be the number of defects that are not detected (defect occurs).

Table 43 – Unambiguity Analysis of ATM System

Scenario	Vague			Subjective			Optional			Weak			Multiple			Implicit			Quantifiable		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1													1			1					
2																3					
3																2					
4																					
5																2					
Total	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**8**	**0**	**0**	**0**	**0**	**0**

Table 44 shows the Completeness analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from ATM System.

Table 44 - Completeness Analysis of ATM System

ID Scenario	Atomicity			Simplicity			Uniformity			Usefulness			Conceptually Soundness			Integrity			Coherency			Uniqueness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1				1			1			1														
2				3	1																			
3														1										
4				1										1										
5				2			1			1				1										
Total	**0**	**0**	**0**	**7**	**1**	**0**	**2**	**0**	**0**	**2**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Table 45 shows the *Consistency* analysis (qualitative) in scenarios and related scenarios from ATM System.

Table 45 - Consistency Analysis of ATM System

ID Scenario	Non-interferential			Boundedness			Reversibility			Liveness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1										1		
2												
3												
4												
5										1		
Total	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	2	**0**	**0**

TITLE: ***ACCESS ATM***
GOAL: User access the ATM.
Context: User wants to use the ATM.
Pre-condition: ATM in ready state for new User
Post-condition: User access granted AND PIN AND card validated.
ACTOR: User
RESOURCE: ATM, card, PIN, account
EPISODES:
1. User inserts card into ATM.
2. ATM asks for a PIN.
3. User types in the numbers of his PIN and presses the Enter button

(Multiplicity - TP: ambiguous indicator)
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)

4. ATM asks for account type.
5. Customer selects account.

(Usefulness - TP: Actor/Resource mentioned in episode is not included in the Actor/Resource element)

6. ATM displays User options.

EXCEPTION:

4.1. ATM rejects unidentifiable card.
(Uniformity - TP: Incomplete Cause)
(Liveness - TP: Never enabled transition)

TITLE: ***CHANGE PIN***
GOAL: User wants to change their PIN.
(Implicit - TP: ambiguous indicator)
Context:
Pre-condition: User already logged onto the ATM
Post-condition: New PIN read to card and Bank account
ACTOR: User
RESOURCE: ATM, PIN
EPISODES:
1. User selects Change PIN.
2. ATM prompts her to enter new PIN.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
3. It enters new PIN.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Missing Subject)
4. ATM prompts User to re-enter new PIN.
5. User re-enters new PIN.
(Simplicity - FP: Contains more than one Subject, Missing Action-Verb)
6. ATM displays New PIN Successful message.
7. ATM displays list of options.
EXCEPTION:
4.1. IF ATM refuses new PIN THEN User asked to re-enter new PIN.
(Simplicity - TP: Missing Action-Verb in Present Tense form)

TITLE: ***CHECK BALANCE***
(Soundness - FP: Title does not describe the Goal)
GOAL: The User wants to check their account balance before withdrawing money.
(Implicit - TP: ambiguous indicator)
Context:
Pre-condition: User already logged onto the ATM.
Post-condition: Balance no longer displayed AND ATM ready for a transaction.
ACTOR: User, Bank
RESOURCE: ATM, account
EPISODES:
1. User selects balance of account.
2. User selects On Screen option.
3. ATM displays current balance on screen.
4. Bank retrieves User's current balance from their account.
(Implicit - TP: ambiguous indicator)
5. ATM prompts for new option.
EXCEPTION:
2.1. IF User selects On Paper option THEN ATM prints balance on receipt.

TITLE: ***MAKE DEPOSIT***
(Soundness - FP: Title does not describe the Goal)
GOAL: The User wants to deposit cash into the ATM
Context:
Pre-condition: The User has logged onto the ATM .
Post-condition: ATM ready for a new transaction.
ACTOR: User
RESOURCE: ATM, envelope, deposit
EPISODES:
1. User selects Deposit.
2. Selects envelope
(Simplicity - TP: Missing Subject)

3. ATM accepts deposit
4. User takes deposit receipt.

EXCEPTION:

3.1. IF ATM rejects deposit envelope THEN ATM signals User of rejection.

TITLE: ***WITHDRAW CASH***

(Soundness - FP: Title does not describe the Goal)

GOAL: User wants to withdraw money.

Context: User wants to use the ATM.

Pre-condition: User has already logged onto the ATM.

Post-condition: ATM ready for next User.

ACTOR: User, Bank

RESOURCE: ATM, account, card

EPISODES:

(*Usefulness - TP: Too long scenario - Num. episodes > 10)*

1. User selects Withdraw Cash.
2. ATM prompts for amount.
3. User enters amount.
4. ATM verifies with the Bank that the User has enough money in account.

(Implicit - TP: ambiguous indicator)

4.1 If insufficient funds in her account,

(Implicit - TP: ambiguous indicator)

(Simplicity - TP: Nested Episode Sentence must be treated by a scenario)

4.2 ATM returns card to User.

4.3 User takes card.

5. ATM releases cash.
6. User takes cash.
7. ATM releases card.
8. User takes card.

(Simplicity - TP: Episode Sentence coincidence with episode "4.3")

EXCEPTION:

7.1. ATM eats card.

(Uniformity - TP: Incomplete Cause)

(Liveness - TP: Never enabled transition)

A1.3
DLibra CRM

Table 46 shows the quantitative analysis in scenarios (Pre-conditions, Post-conditions, Episodes, Exceptions) from Online Broker System, and how they are mapped in Petri-Net elements (Input places, Output places, Transitions).

Table 46 - Quantitative Analysis of Online Broke System

ID Scenario	Scenario	Num. Pre-conditions/Conditions/ Causes/Constraints	Num. Post-conditions	Num. Episodes	Num. Exceptions	Num. Input Places - Petri-Net	Num. Transitions - Petri-Net	Num. Output Places - Petri-Net	Num. Dummy Places - Petri-Net
1	Add a new client	4	0	4	3	4	7	0	9
2	Add a new contract	3	0	6	2	3	8	0	10
3	Add a new installation	3	0	6	2	3	8	0	10
4	Browse clients	1	0	6	1	1	7	0	9
5	Browse information	1	0	6	1	1	7	0	9
6	Delete client	1	0	4	1	1	5	0	7
7	Delete contract	1	0	4	1	1	5	0	7
8	Delete installation	1	0	4	1	1	5	0	7
9	Edit client data	4	0	4	3	4	7	0	9
10	Edit contract	3	0	6	2	3	8	0	10
11	Edit installation	3	0	6	2	3	8	0	10
12	Log in to the system	2	0	4	1	2	5	0	7
13	Prepare a report	1	0	7	1	1	8	0	10
14	Request for licence	3	0	8	3	3	11	0	13
15	Search	2	0	5	2	2	7	0	9
Total		**33**	**0**	**80**	**26**	**33**	**106**	**0**	**136**

Table 47 shows the Unambiguity analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Dlibra System. Let ***TP*** be the number of defects detected correctly by out analysis approach; ***FP*** be the number of defects detected incorrectly (defect does not occur); ***FN*** be the number of defects that are not detected (defect occurs).

Table 47 – Unambiguity Analysis of Dlibra System

ID Scenario	Vague			Subjective			Optional			Weak			Multiple			Implicit			Quantifiable		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1													1			1			1		
2										1			1			3			1		
3										1			1			3					
4										1			1								
5										3			2								
6													1								
7													1								
8													1								
9													1			1					
10																1					
11																1					
12																			1		
13													1	2							
14	1												1			1			1		
15																1					
Total	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**6**	**0**	**0**	**12**	**2**	**0**	**12**	**0**	**0**	**4**	**0**	**0**

Table 48 shows the Completeness analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Dlibra System. Consistency is not shown because we do not have a baseline manually identified by Requirements Engineers.

Table 48 - Completeness Analysis of Dlibra System

ID Scenario	Atomicity			Simplicity			Uniformity			Usefulness			Conceptually Soundness			Integrity			Coherency			Uniqueness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1				2																				
2				1																				
3				1																				
4				5																				
5				5																				
6				3																				
7				3																				
8				3																				
9				2																				
10				1																				
11				1																				
12																								
13				3																				
14				5																				
15	1													1										
Total	**1**	**0**	**0**	**35**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

TITLE: ***Add a new client***
GOAL: Add a new client
CONTEXT: user Add a new client
ACTOR: User, System
RESOURCE: database, clients
EPISODES:
1. User select option for adding new clients.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User fills all required personal client data forms.
(Quantifiable - TP: ambiguous indicator)
3. System verifies correctness of data.
4. System adds a new client to the database and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
3.1. IF Data is incomplete or incorrect THEN System informs user about problems.
3.2. IF Client with same personal data already exists THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)
4.1. IF Client can't be added THEN System informs user about reason why client can't be added.

TITLE: ***Add a new contract***
GOAL: Add a new contract
CONTEXT: Add a new contract
ACTOR: User, System
RESOURCE: client, contract
EPISODES:
1. User select a client for whom new contract will be added.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Implicit - TP: ambiguous indicator)
(Weakness - TP: ambiguous indicator)
2. User chooses option for adding new contract.
3. System displays transaction form.
4. User fills all required data.
(Quantifiable - TP: ambiguous indicator)
5. System verifies information.
6. System saves contract and bounds it to the selected client.

(Multiplicity - TP: ambiguous indicator)
(Implicit - TP: ambiguous indicator)
EXCEPTION:
5.1. IF Data is incomplete or incorrect THEN System informs user about problems.
6.1. IF contract can't be saved THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)

TITLE: ***Add a new installation***
GOAL: Add a new installation
CONTEXT: Add a new installation
ACTOR: User, System
RESOURCE: client, installation
EPISODES:
1. User select a client for whom new installation will be added.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Implicit - TP: ambiguous indicator)
(Weakness - TP: ambiguous indicator)
2. User chooses option for adding new installation.
3. System displays installation form.
4. User fills required data.
5. System verifies information.
6. System saves installation and bounds it to the selected client.
(Multiplicity - TP: ambiguous indicator)
(Implicit - TP: ambiguous indicator)
EXCEPTION:
5.1. IF Data is incomplete or incorrect THEN System informs user about problems.
6.1. IF Installation can't be saved THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)

TITLE: ***Browse clients***
GOAL: Browse clients
CONTEXT: Browse clients
ACTOR: user, System
RESOURCE: clients, scenario
EPISODES:
1. User select option for browsing clients.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. System displays the list of clients.
3. User may filter clients with specified criteria.
(Simplicity - TP: Missing Action-Verb in Present Tense) (Weakness: ambiguous indicator)
4. User may sort clients.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Weakness - TP: ambiguous indicator)
5. User may view details about selected client.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Weakness - TP: ambiguous indicator)
6. Scenario ends when user logs out or selects different option.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
2.1. IF There are no clients to display THEN System displays blank list.

TITLE: ***Browse information***
GOAL: Browse information
CONTEXT: Browse information
ACTOR: User, System
RESOURCE: licences, keys, contracts, installations, scenario
EPISODES:
1. User select option for browsing data.
(Simplicity - TP: Missing Action-Verb in Present Tense)

2. System displays the list of licences, keys, contracts and installations (grouping by type).
(Multiplicity - TP: ambiguous indicator)
3. User may filter data with specified criteria.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Weakness - TP: ambiguous indicator)
4. User may sort data.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Weakness - TP: ambiguous indicator)
5. User may view details about selected element.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Weakness - TP: ambiguous indicator)
6. Scenario ends when users logs out or select different option.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
2.1. IF There are no data to display THEN System displays blank list.

TITLE: ***Delete client***
GOAL: Delete client
CONTEXT: Delete client
ACTOR: User , System
RESOURCE: client, clients, database
EPISODES:
1. User select option for deleting clients.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User delete chosen client.
(Simplicity - TP: Missing Action-Verb in Present Tense)
3. System verifies possibility to perform deleting.
4. System saves changes to the database and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
3.1. IF Client can not be deleted THEN System informs user about the conditions.

TITLE: ***Delete contract***
GOAL: Delete contract
CONTEXT: Delete contract
ACTOR: User, System
RESOURCE: contract, database
EPISODES:
1. User select option for deleting contract.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User delete chosen contract.
(Simplicity - TP: Missing Action-Verb in Present Tense)
3. System verifies possibility to perform deleting.
4. System saves changes to the database and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
3.1. IF Contract can not be deleted THEN System informs user about problems.

TITLE: ***Delete installation***
GOAL: Delete installation
CONTEXT: Delete installation
ACTOR: User, System
RESOURCE: installation, database
EPISODES:
1. User select option for deleting installation.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User delete chosen installation.

(Simplicity - TP: Missing Action-Verb in Present Tense)
3. System verifies possibility to perform deleting.
4. System saves changes to the database and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
3.1. IF Installation can not be deleted THEN System informs user about problems.

TITLE: ***Edit client data***
GOAL: Edit client data
CONTEXT: Edit client data
ACTOR: user, System
RESOURCE: clients, database
EPISODES:
1. User select option for editing clients.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User modifies personal client data.
3. System verifies correctness of data.
4. System saves a new client data to the database and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
3.1. IF Data is incomplete or incorrect THEN System informs user about problems.
3.2. IF Client with same personal data already exists THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)
4.1. IF Client data changes can't be saved THEN System informs user about reason why client can't be modified.

TITLE: ***Edit contract***
GOAL: Edit contract
CONTEXT: Edit contract
ACTOR: User, System
RESOURCE: client, contract
EPISODES:
1. User select a client.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User chooses option for editing an existing contract.
3. System displays transaction form.
4. User changes desired data.
5. System verifies information.
6. System saves changed contract.
EXCEPTION:
5.1. IF Data is incomplete or incorrect THEN System informs user about problems.
6.1. IF contract can't be saved THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)

TITLE: ***Edit installation***
GOAL: Edit installation
CONTEXT: Edit installation
ACTOR: User, System
RESOURCE: client, installation
EPISODES:
1. User select a client.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User chooses option for editing an existing installation.
3. System displays installation form.
4. User changes desired data.
5. System verifies information.
6. System saves changed installation.
EXCEPTION:

5.1. IF Data is incomplete or incorrect THEN System informs user about problems.
6.1. IF Installation can't be saved THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)

TITLE: ***Log in to the system***
GOAL: Log in to the system
CONTEXT: User Log in to the system
ACTOR: User, System
RESOURCE: main page, login option
EPISODES:
1. User selects login option.
2. User provides all required data.
(Quantifiable - TP: ambiguous indicator)
3. System verifies correctness of data.
4. System displays a main page.
EXCEPTION:
3.1. IF Data is incomplete or incorrect THEN System asks for data again.

TITLE: ***Prepare a report***
GOAL: Prepare a report
CONTEXT: Prepare a report
ACTOR: User, System
RESOURCE: report, database, file
EPISODES:
1. User select option for creating reports.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. System displays a list of possible fields in the report.
3. User selects fields to be included in the report and rules to filter values from database.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
4. User order report generation.
(Simplicity - TP: Missing Action-Verb in Present Tense)
5. System asks for type and localisation of the output file with report.
(Multiplicity - FP: ambiguous indicator)
6. User selects the type and localisation of the output file with report.
(Multiplicity - FP: ambiguous indicator)
7. System generates a report.
EXCEPTION:
7.1. IF Report can't be saved in given location THEN System displays information.

TITLE: ***Request for licence***
GOAL: Request for licence
CONTEXT: Request for licence
ACTOR: User, System, PCSS Team Participant
RESOURCE: dLibra server, licence, contracts, file
EPISODES:
1. User select option for requesting a new licence.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. System displays the list of user's contracts.
3. User selects one contract for licence request.
4. System contact with dLibra server to obtain all necessary data.
(Simplicity - TP: Missing Action-Verb in Present Tense)
(Quantifiable - TP: ambiguous indicator)
(Vagueness - TP: ambiguous indicator)
5. System validate given data.
(Simplicity - TP: Missing Action-Verb in Present Tense)
6. System store a request new licence and informs user about it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
7. PCSS Team Participant approve request for a new licence.

(Simplicity - TP: Missing Action-Verb in Present Tense)
8. User downloads the licence file.
EXCEPTION:
2.1. IF There are no contracts THEN System displays blank list.
3.1. IF selected contract can not have more licenses THEN System informs user about that fact.
(Implicit - TP: ambiguous indicator)
5.1. IF Data is not valid THEN System informs user about incorrect data.

TITLE: ***Search***
(Atomicity - TP: Missing Object)
(Soundness -FP: Title does not describe the Goal)
GOAL: Search
CONTEXT: Search
ACTOR: User, System
RESOURCE: filter, criteria, database, results
EPISODES:
1. User select option for searching.
(Simplicity - TP: Missing Action-Verb in Present Tense)
2. User selects subject of search (clients, contracts, installations).
3. System displays list of possible criteria.
4. User creates filter for searching.
(Simplicity - TP: Missing Object)
5. System search the database and displays the results.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
4.1. IF Chosen criteria are invalid THEN System warns user.
5.1. IF No records found THEN System displays blank list.

A1.4
Mobile News

Table 49 shows the quantitative analysis in scenarios (Pre-conditions, Post-conditions, Episodes, Exceptions) from Mobile News System, and how they are mapped in Petri-Net elements (Input places, Output places, Transitions).

Table 49 - Quantitative Analysis of Mobile News System

ID Scenario	Scenario	Num. Pre-conditions/Conditions/ Causes/Constraints	Num. Post-conditions	Num. Episodes	Num. Exceptions	Num. Input Places - Petri-Net	Num. Transitions - Petri-Net	Num. Output Places - Petri-Net	Num. Dummy Places - Petri-Net
1	Add a new channel	0	0	12	1	0	13	0	15
2	Add a new channel group	0	0	7	0	0	7	0	9
3	Configure the server	0	0	4	1	0	5	0	7
4	Configure user preferences	0	0	4	0	0	4	0	6
5	Delete a channel	0	0	10	0	0	10	0	12
6	Delete a channel group	0	0	7	0	0	7	0	9
7	Delete a user group	0	0	7	1	0	8	0	10
8	Delete news	0	0	2	0	0	2	0	4
9	Download news	0	0	6	0	0	6	0	8
10	Post a group message	0	0	4	0	0	4	0	6
11	Read news	0	0	6	0	0	6	0	8
12	Register a new user	1	0	5	2	1	7	0	9
13	Run the application	0	0	3	0	0	3	0	5
14	Subscribe/unsubscribe news channels	0	0	7	0	0	7	0	9
15	Update news	0	0	5	0	0	5	0	7
Total		1	0	89	5	1	94	0	124

Table 50 shows the Unambiguity analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Mobile News System. Let ***TP*** be the number of defects detected correctly by out analysis approach; ***FP*** be the number of defects detected incorrectly (defect does not occur); ***FN*** be the number of defects that are not detected (defect occurs).

Table 50 – Unambiguity Analysis of Mobile News System

Scenario	Vague			Subjective			Optional			Weak			Multiple			Implicit			Quantifiable		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1	1												5			1			1		
2													4								
3													1								
4	1												1								
5													3			2			2		
6													2			1			1		
7													1						1		
8													1			1			1		
9													2			1			2		
10													1								
11	1												1			1					
12													1			4					
13																					
14	2												4			2					
15	1													1							
Total	6	0	0	0	0	0	0	0	0	0	0	0	27	1	0	13	0	0	8	0	0

Table 51 shows the Completeness analysis (qualitative) in scenarios (Title, Goal, Episodes, Exceptions) from Mobile News System. Consistency is not

shown because we do not have a baseline manually identified by Requirements Engineers.

Table 51 - Completeness Analysis of Mobile News System

ID Scenario	Atomicity			Simplicity			Uniformity			Usefulness			Conceptually Soundness			Integrity			Coherency			Uniqueness		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
1				4			1			1														
2				2																				
3				1			1																	
4				2																				
5				2						1														
6				1																				
7					1		1																	
8										1														
9				1																				
10				1																				
11				1																				
12				5				1																
13				1																				
14				5		1																		
15						1																		
Total	0	0	0	26	1	2	3	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TITLE: ***Add a new channel***
GOAL: Add a new channel
CONTEXT: Add a new channel
ACTOR: Administrator, System
RESOURCE: channel, service, channel group, database
EPISODES:
(*Usefulness - TP: Too long scenario - Num. episodes > 10)*
1. Administrator logs on to the administration panel.
2. System displays administration options.
3. Administrator selects the Group and channel management option.
(Multiplicity - TP: ambiguous indicator)
4. System displays a list of defined channel groups and an add/delete group menu.
(Multiplicity - TP: ambiguous indicator)
5. Administrator chooses a group to which he wants to add a new channel.
(Simplicity - TP: Contains more than one Action-Verb)
(Implicit - TP: ambiguous indicator)
6. System displays a list of channels in the selected group and an add/delete menu.
(Multiplicity - TP: ambiguous indicator)
7. Administrator types the name of the channel and the URL of the news service and selects Add channel.
(Simplicity - TP: Contains more than one Action-Verb)
(Multiplicity - TP: ambiguous indicator)
8. System checks if a channel with the given name or URL has not been already defined and if so, inserts the channel information into a database.
(Multiplicity - TP: ambiguous indicator)
9. System adds an information about the new channel to a group message.
10. See step 6.
(Simplicity - TP: Missing Subject)
11. Administrator selects the Finish option.
12. System posts a group message containing information about all new channels in the selected channel group.
(Quantifiable - TP: ambiguous indicator)
EXCEPTION:
5.1. Administrator adds more channels. Proceed to step 7.
(Simplicity - TP: Contains more than one Sentence)
(Uniformity - TP: Incomplete cause)
(Vagueness - TP: ambiguous indicator)

TITLE: ***Add a new channel group***
GOAL: Add a new channel group
CONTEXT: Add a new channel group

ACTOR: Administrator, System
RESOURCE: Group, channel, database
EPISODES:
1. Administrator logs on to the administration panel.
2. System displays administration options.
3. Administrator selects the Group and channel management option.
(Multiplicity - TP: ambiguous indicator)
4. System displays a list of defined channel groups and an add/delete group menu.
(Multiplicity - TP: ambiguous indicator)
5. Administrator types the name of a new group and selects Add group.
(Simplicity - TP: Contains more than one Action-Verb)
(Multiplicity - TP: ambiguous indicator)
6. 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.
(Multiplicity - TP: ambiguous indicator)
7. See step 4.
(Simplicity - TP: Missing Subject)

TITLE: ***Configure the server***
GOAL: Configure the server
CONTEXT: Configure the server
ACTOR: Administrator
RESOURCE: configuration settings
EPISODES:
1. Administrator logs on to the administration panel.
2. Administrator selects the Configure option.
3. Administrator chooses and changes the desired settings.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
4. Administrator saves configuration settings.
EXCEPTION:
4.1. Administrator cancels configuration changes.
(Uniformity - TP: Incomplete cause)

TITLE: ***Configure user preferences***
GOAL: Configure user preferences
CONTEXT: Configure user preferences
ACTOR: User, System
RESOURCE: Preferences, options
EPISODES:
1. User chooses the Preferences option.
2. System displays a list of available options (i.e. font and color settings, local news caching, etc..).
(Vagueness - TP: ambiguous indicator)
3. User configures the option according to his/her preferences and confirm the changes.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
4. System saves user preferences configuration and displays main application view.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)

TITLE: ***Delete a channel***
GOAL: Delete a channel
CONTEXT: Delete a channel
ACTOR: Administrator, System
RESOURCE: channel, channels, group, database
EPISODES:
(*Usefulness - TP: Too long scenario - Num. episodes > 10)*
1. Administrator logs on to the administration panel.
2. System displays administration options.
3. Administrator selects the Group and channel management option.

(Multiplicity - TP: ambiguous indicator)
4. System displays a list of defined channel groups and an add/delete group menu.
(Multiplicity - TP: ambiguous indicator)
5. Administrator chooses a group containing the channel he wants to delete.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
6. System displays a list of channels in the selected group and an add/delete menu.
(Multiplicity - TP: ambiguous indicator)
7. Administrator selects the channel(s) he wants to delete and chooses the Delete option.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
8. System deletes the selected channels from the database.
9. System posts a group message containing information about the deleted channels in the selected channel group to all users involved (subscribing the deleted channels).
(Quantifiable - TP: ambiguous indicator)
10. System deletes all subscription information concerning the deleted channels.
(Quantifiable - TP: ambiguous indicator)

TITLE: ***Delete a channel group***
GOAL: Delete a channel group
CONTEXT: Delete a channel group
ACTOR: Administrator, System
RESOURCE: channel group
EPISODES:
1. Administrator logs on to the administration panel.
2. System displays administration options.
3. Administrator selects the Group and channel management option.
(Multiplicity - TP: ambiguous indicator)
4. System displays a list of defined channel groups and an add/delete group menu.
(Multiplicity - TP: ambiguous indicator)
5. Administrator selects the group(s) he wants to delete and chooses the Delete option.
(Simplicity - TP: Contains more than one Action-Verb)
6. System asks for confirmation.
(Implicit - TP: ambiguous indicator)
7. System deletes all channels from the selected groups (see: UC5, steps 8 to 10).
(Quantifiable - TP: ambiguous indicator)

TITLE: ***Delete a user group***
GOAL: Delete a user group
CONTEXT: Delete a user group
ACTOR: Administrator, System
RESOURCE: users, accounts
EPISODES:
1. Administrator logs on to the administration panel.
2. System displays administration options.
3. Administrator selects the Delete users option.
4. System displays the users deletion menu.
5. Administrator selects deletion options (i.e. date of users` last login).
6. Administrator confirms deletion request.
7. System finds all users matching deletion criteria and deletes found user accounts.
(Simplicity - FN: Contains more than one Action-Verb)
(Quantifiable - TP: ambiguous indicator)
(Multiplicity - TP: ambiguous indicator)
EXCEPTION:
6.1. Administrator cancels user deletion.
(Uniformity - TP: Incomplete cause)

TITLE: ***Delete news***
GOAL: Delete news
CONTEXT: Delete news

ACTOR: System
RESOURCE: messages, database
EPISODES:
(*Usefulness - TP: Too short scenario - Num. episodes < 3)*
1. System queries the database for news messages, whose expiry date and time have passed.
(Implicit - TP: ambiguous indicator)
(Multiplicity - TP: ambiguous indicator)
2. System deletes all returned messages from the database.
(Quantifiable - TP: ambiguous indicator)

TITLE: ***Download news***
GOAL: Download news
CONTEXT: Download news
ACTOR: User, system
RESOURCE: news, server, messages, channels, database
EPISODES:
1. User chooses to update locally stored news.
2. System sends a HTTP request to the Mobile News server.
3. Server sends all pending group messages.
(Quantifiable - TP: ambiguous indicator)
4. Server sends separate news messages from all subscribed channels.
(Quantifiable - TP: ambiguous indicator)
5. System receives news messages and stores them in a local database.
(Multiplicity - TP: ambiguous indicator)
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
6. System displays a list of groups with subscribed channels and the number of new messages in each of them.
(Multiplicity - TP: ambiguous indicator)

TITLE: ***Post a group message***
GOAL: Post a group message
CONTEXT: Post a group message
ACTOR: Administrator, User
RESOURCE: message, data
EPISODES:
1. Administrator logs on to the administration panel.
2. Administrator selects the Post group message option.
3. Administrator types the message and posts it.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
4. User receives the message when downloading new data.

TITLE: ***Read news***
GOAL: Read news
CONTEXT: Read news
ACTOR: User, System
RESOURCE: messages, preferences, hyperlink
EPISODES:
1. User chooses a news group from the Today menu.
2. System displays a list of topics of available messages in chosen group.
(Vagueness - TP: ambiguous indicator)
3. User chooses a topic.
4. System displays the message using user's appearance preferences.
5. User reads the message and closes it or uses a hyperlink to go to the full message.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
(Implicit - TP: ambiguous indicator)
6. System marks the message as Read.

TITLE: ***Register a new user***
GOAL: Register a new user
CONTEXT: Register a new user
ACTOR: User, System
RESOURCE: server, user account, ID , preferences
EPISODES:
1. System asks the user if he/she wants to register.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
2. User confirms he/she wants to register.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
3. System sends a registration request to the server.
4. Server creates a new user account and sends back a user ID.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
5. System stores the user ID and instructs the user how to subscribe news channels or configure his/her preferences.
(Implicit - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
EXCEPTION:
2.1. IF User refuses to register THEN System displays an information that it cannot be used without prior registration.
(Implicit - TP: ambiguous indicator)
2.1.2 User confirms the message.
(Simplicity - TP: Nested Exception Solution must be treated by a scenario)
(Uniformity - FP: Incomplete cause)

TITLE: ***Run the application***
GOAL: Run the application
CONTEXT: Run the application
ACTOR: User, System
RESOURCE: application, channels
EPISODES:
1. User starts the application.
2. System checks for registration information.
3. IF the user is already registered THEN the system automatically updates news messages from subscribed channels (refer to Download news use case). If no, the system attempts to Register a new user (refer to Register a new user use case).
(Simplicity - TP: Contains more than one Action-Verb)

TITLE: ***Subscribe/unsubscribe news channels***
GOAL: Subscribe/unsubscribe news channels
CONTEXT: Subscribe/unsubscribe news channels
ACTOR: User, System
RESOURCE: server, channels, database
EPISODES:
1. User chooses the Channel subscription option.
2. System requests for and downloads a list of available groups and channels.
(Multiplicity - TP: ambiguous indicator)
(Vagueness- TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
3. System displays a tree view of available groups and channels and marks those already subscribed by the user.
(Multiplicity - TP: ambiguous indicator)
(Vagueness- TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
4. User selects the channels he/she wants to subscribe and/or deselects already subscribed channels to unsubscribe them and chooses the Change subscription options.
(Implicit - TP: ambiguous indicator)

(Simplicity - TP: Contains more than one Action-Verb)
5. System sends the subscription configuration to the Mobile News server and waits for confirmation.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
6. Server alters the user's subscription configuration in a database and sends a change confirmation.
(Multiplicity - TP: ambiguous indicator)
(Simplicity - TP: Contains more than one Action-Verb)
7. System receives the confirmation and displays it.
(Implicit - TP: ambiguous indicator)
(Simplicity - FN: Contains more than one Action-Verb)

TITLE: ***Update news***
GOAL: Update news
CONTEXT: Update news
ACTOR: Daemon, System
RESOURCE: service, file, message, database
EPISODES:
1. Daemon sends a HTTP request to a defined news service.
2. System receives a RSS-like formatted news file.
3. System parses the received news file.
4. System assigns an expiry date and time to each incoming message.
(Simplicity - FP: Contains more than one Action-Verb)
(Multiplicity - FP: ambiguous indicator)
5. System inserts appropriate parts of the news file into a news database.
(Vagueness - TP: ambiguous indicator)

Appendix A2
Quality Models of Related Work

A.2.1.
Static Analysis of Software Requirements Specification

A.2.1.1.
Static Analysis of Requirement Statements

Table 52 - Quality Indicators of ARM (Wilson et al., 1997)

Property	Description	Indicators
Imperatives	The sentence contains words or phrases that command that something must be provided.	shall, must, must not, is required to, are applicable.
Continuances	The sentence contains phrases that follow an imperative and introduce the SRS at a lower level.	below: , as follows: , following: , listed::
Directives	The sentence contains words or phrases that point to illustrative information within the SRS.	figure, table, for example, note:
Options	The sentence contains words that give the developer latitude in satisfying the specification statements that contain them.	can , may, optionally
Weak Phrases	The sentence contains clauses that are apt to cause uncertainty and leave room for multiple interpretations.	adequate, as a minimum, as applicable, easy, as appropriate
Readability	It measures the difficulty in reading the Document or a sentence. This metric is the Coleman-Liau Formula readability metric: (5.89* letters/words-0.3*sentences/(100*words)-15.8])	If it is > 55.8 the document is difficult -to-read.

Table 53 - Expressiveness Quality Model of QuARS (Gnesi et al., 2005)

Property	Description	Indicators
Vagueness	The sentence contains words or phrases having a non uniquely quantifiable meaning.	clear, easy, strong, good, bad, efficient, useful, significant
Subjectivity	The sentence contains words or phrases expressing personal opinions or feeling.	similar, better, similarly, worse, having in mind, take into account
Optionality	The sentence contains words or phrases expressing an optional part (i.e. a part that can or cannot be considered).	possibly, eventually, if case, if possible, if appropriate, if needed
Implicitly	The sentence does not specify the subject or object by means of its specific name but uses pronoun or other indirect reference.	this, these, that, those, it, they, previous, next, following, below
Weakness	The sentence contains a weak verb. A verb that makes the sentence not imperative is considered weak.	can, could, may, …
Under-specification	The sentence contains a word identifying a class of objects without a modifier specifying an instance of this class	flow instead of data flow, control flow, .. , testing instead of functional testing, unit testing
Multiplicity	The sentence has more than one main verb, subject or object	and, or, and/or, …
Readability	It measures the difficulty in reading the Document or a sentence. This metric is the Coleman-Liau Formula readability metric: (5.89* letters/words-0.3*sentences/(100*words)-15.8])	If it is >15 the document is difficult -to-read.

Table 54 - Ambiguity Indicators of SRRE (Tjong, 2008)

Property	Description	Indicators
Continuance	The sentence introduces further specification	as follows, below, following, in addition, in particular
Coordinator	The sentence introduces a coordination ambiguity	and, and/or, or
Directive	The sentence introduces extra information	e.g., etc., figure, for example, i.e., note, table
Incomplete	The sentence introduces information that are not in SRS	as a minimum, as defined, as specified, in addition, is defined, no practical limit
Optional	The sentence expresses an optional part	as desired, at last, either, eventually, if appropriate
Pronoun	The sentence uses pronouns or indirect reference	anyone, anybody, anything i, it, its, itself, me, mine, most, my, myself, nobody, none, no one, nothing, our, ours, ourselves, she, someone, somebody, yourselves
Plural	The sentence contains plural words	The Plural corpus contains a list of 11,287 plural nouns, each ending in s
Quantifier	The sentence introduces terms used for quantification	all, any, few, little, many, much, several, some
Vague	The sentence introduces terms that contribute vagueness	/,< >, ( ), [ ], { }, ;, ?, !, adaptability, additionally, adequate, aggregate, also, ancillary, arbitrary, appropriate, as appropriate varying
Weak	The sentence contains a weak verb	can, could, may, might, ought to, preferred, should, will, would

Table 55 - Requirements language criteria (IEEE, 2011; Femmer et al., 2014)

Smell	Description	Indicators
Ambiguous Adverbs and Adjectives	Refer to adverbs and adjectives that are unspecific	almost always, significant, minimal
Vague Pronouns	Are unclear relations of a pronoun	*Using Part-of-speech Tagging*
Subjective Language	Refer to words of which the semantics is not objective	User friendly, easy to use, cost effective
Comparative Phrases	Are used in requirements that express a relation of the system to specific other systems	*Using Morphological Analysis*
Superlatives	Are used in requirements that express a relation of the system to all other systems	*Using Morphological Analysis and Part-of-speech Tagging*
Negative Statements	Are "statements of system capability not to be provided"	must not
Open-ended, Non-verifiable Terms	Are hard to verify as they offer a choice of possibilities	Provide support, but not limited to, as a minimum
Loopholes	Enable stakeholders to ignore certain parts of the application	Is possible, as appropriate, as applicable
Incomplete References	Are references that a reader cannot follow	*Not implemented*

Table 56 - Potentially problematic constructs (from Berry et al., 2012)

Smell	Description	Indicators
Warn_AND	The "*and*" conjunction can imply several meanings,	and
Warn_OR	The "*or*" conjunction can imply "exclusive or", or "inclusive or".	or
Warn_Quantifier	Terms used for quantification	all, any, every
Warn_Pronoun	Pronouns can lead to referential ambiguity.	they
Warn_VagueTerms	There are several vague terms that are commonly used in requirements documents.	user-friendly, support, acceptable, up to, periodically
Warn_PassiveVoice	Passive voice blurs the actor of the requirement and must be avoided in requirements.	it
Warn_Complex_Sentence	Using *multiple conjunctions* in the same requirements sentence make the sentence hard to read and are likely to cause ambiguity.	and, or
Warn_Plural_Noun	*Plural Nouns* can potentially lead to ambiguous situations	
Warn_Adverb_in_Verb_Phrase	*Adverbial verb phrases* are discouraged due to vagueness and the chances of important details remaining tacit in the adverb	periodically
Warn_Adj_followed_by_Conjunction	The adjective followed by two nouns separated by a conjunction, can lead to ambiguity due to the possible relation of adjective with just first noun or both nouns.	compliant

Table 57 - Quality User Story Framework (Lucassen et al., 2015)

Quality	Criteria	Description
Syntactic: quality, concerning the textual structure of a user story without considering its meaning.	*Atomic*	A user story expresses a requirement for exactly one feature
	Minimal	A user story contains nothing more than role, means and ends
	Well-formed	A user story includes at least a role and a means
Semantic: quality, concerning the relations and meaning of (parts of) the user story text.	Conflict-free	A user story should not be inconsistent with any other user story
	Conceptually sound	The means expresses a feature and the ends expresses a rationale, not something else
	Problem-oriented	A user story only specifies the problem, not the solution to it
	Unambiguous	A user story avoids terms or abstractions that may lead to multiple interpretations
Pragmatic: quality, regarding choosing the most effective alternatives for communicating a given set of requirements.	Complete	Implementing a set of user stories creates a feature-complete application, no steps are missing
	Explicit dependencies	Link all unavoidable, non-obvious dependencies on user stories
	Full sentence	A user story is a well-formed full sentence
	Independent	The user story is self-contained, avoiding inherent dependencies on other user stories
	Scalable	User stories do not denote too coarse-grained requirements that are difficult to plan and prioritize
	Uniform	All user stories follow roughly the same template
	Unique	Every user story is unique, duplicates are avoided

A.2.1.2. Static Analysis of Scenarios

Table 58 - Taxonomy of defects in use case models (Anda and Sjoberg, 2002)

Checklist Element		Description
Use case diagram: check the completeness and consistency in use case diagrams	Actor	Human users or external entities that will interact with the system are not identified. Incorrect description of actors or wrong connection between actor and use case. Description of actor is inconsistent with its behavior in use cases. Too broadly defined actors or ambiguous description of actor. Actors that do not derive value from/provide value to the system.
	Use case	Required functionality is not described in use cases. Actors have goals that do not have corresponding use cases. Incorrect description of a use case Description is inconsistent with reaching the goal of the use case. Name of use case does not reflect the goal of the use case. Use cases with functionality outside the scope of the system or use cases that duplicate functionality.
Use case description: check the completeness and consistency in use cases and their relationships	Flow of events	Input or output for use cases is not described. Events that are necessary for understanding the use cases are missing. Incorrect description of one or several events. Events that are inconsistent with reaching the goal of the use case they are part of. Ambiguous description of events, perhaps because of too little detail. Superfluous steps or too much detail in steps.
	Variations	Variations that may occur when attempting to achieve the goal of a use case are not specified. Incorrect description of a variation. Variations that are inconsistent with the goal of the use case. Ambiguous description of what leads to a particular variation. Variations that are outside the scope of the system.
	Relation between use cases	Common functionality is not separated out in included use cases. Inconsistencies between diagram and descriptions, inconsistent terminology, inconsistencies between use cases, or different level of granularity.
	Trigger, pre-condition and post-condition	Trigger, pre- or post-conditions have been omitted. Incorrect assumptions or results have led to incorrect pre- or post- conditions. Pre- or post- conditions are inconsistent with goal or flow of events. Ambiguous description of trigger, pre- or post-condition Superfluous trigger, pre-or post-conditions.

Table 59 - Scenario Checklist (Leite et al., 2000; Leite et al., 2005)

Checklist Element		Description
Intra-scenario: verify each component in every scenario to confirm its consistency with the components and adherence to the scenario model	Syntactic verification	Check the existence of more than one episode per scenario; Check the syntax of each scenario element as established in the scenario model;
	Relationship among components	Check that every Actor participates in at least one episode; Check that every Actor mentioned in episodes is included in the Actor element; Check that every Resource is used in at least one episode; Check that every Resource mentioned in episodes is included in the Resource element;
	Semantic verification	Check the coherence between the Title and the Goal; Ensure that the set of Episodes satisfies the Goal and is within the Context; Ensure that actions presents in the Pre-conditions are already performed; Ensure that Episodes contain only actions to be performed;
Inter-scenario: check the relationship among different scenarios looking for overlaps or gaps	Scenario relationship	Check that every Episode identified as sub-scenario exists within the set of scenarios; Check that the set of Episodes of every sub-scenario is not already included in another scenario; Check that every Exception is treated by a scenario; Check that every Pre-condition is either an uncontrollable fact or is satisfied by another scenario; Check coherence between related scenario Pre-conditions and scenario Pre-conditions; Check that geographical and Temporal location of related scenarios are equal or more restricted than those of scenario;
	Scenario overlap	Check that Goal coincidence only takes place in different situations; Check that Episode coincidence only takes place in different situations; Check that Context coincidence only takes place in different situations;
LEL Coverage: ensure that LEL symbols are properly used and that every phrase emphasized as a LEL symbol is actually part of LEL.	Check that every lexicon symbol is identified; Check the correct use of lexicon symbols; Check that Actors are preferentially Subject symbols; Check that Resources are preferentially Object symbols; Check that the behavioral response of Subject symbols are covered by scenarios;	

Table 60 - The 7Cs Verification Heuristics (Phalp et al., 2007)

Property		**Description**
Coverage	Scope	The use case should contain all that is required to answer the problem.
	Span	The use case should only contain detail relevant to the problem statement. Extra unnecessary information provided is out of problem scope and not required.
Cogent	Text Order	The use case should follow a logical path with events in the description in the correct order.
	Dependencies	The use case should complete as an end-to-end transaction (which can include alternative/exceptional flows). Does the actor reach a state that stops the transaction from terminating as expected?
	Rational Answer	The logic of the use case description should provide a plausible answer to the problem.
Coherent	The sentence being written should repeat a noun in the last sentence or a previous sentence, if possible. The description is easier to read and quicker to understand if there is logical coherence throughout.	
Consistent Abstraction	The use case should be at a consistent level of abstraction through- out. Mixing abstraction levels (problem domain, interface specification, internal design mixes) may cause difficulty in understanding.	
Consistent Structure	Variations	Alternative paths should be excluded from the main flow. Inclusion of alternative paths in the main flow reduces readability.
	Sequence	Numbering of events in the main flow should be consistent.
Consistent Grammar	Simple present tense should be used throughout. Adverbs, adjectives, pronouns, synonyms and negatives should be avoided.	
Consideration of Alternatives	Separation	There should be a separate section for any alternative/exceptional paths to the main flow.
	Viable	Alternatives should be viable and make sense.
	Numbering	Alternative numberings should exactly match the numbers in the main flow.

Table 61 - The Use Case Defects (Ciemniewska and Jurkiewicz, 2007)

Level	Defect	Description
Specification-Level Bad Smells	*Use-Case Duplication*	At the level of requirements specification, where there are many use cases, a quite common defect which we have observed is *Use-Case Duplication.*
Use-Case Level	Too long or too short use cases	It is strongly recommended to keep use cases 3-9 steps long. Too long use cases are difficult to read and understand. Too short use cases, consisting of one or two steps, distract a reader from the context and, as well, make the specification more difficult to understand.
	Complicated extension	When the interruption causes the execution of a repeatable, consistent sequence of steps, then this sequence should be extracted to a separate use case
	Repeated actions in neighbouring steps	Every step of a use case should represent one particular action. The action may consist of one or more moves which can be taken as an integrity. Every step should contain significant information which rather reflect user intent then a single move. Splitting these movements into separate steps may lead to long use cases, bothersome to read and hard to maintain.
	Inappropriate naming	Every use case should have a descriptive name. The title of each use case presents a goal that the primary actor wants to achieve. There is a few conventions of naming use cases, but it is preferable to use active verb phrase in the use case name. Furthermore, chosen convention should be used consistently in all us cases.
Step Level	Too Complex Sentence Structure	The structure of a sentence used for describing each step of use case should be as simple as possible. It means that it should generally consists of a subject, a verb, an object and a prepositional phrase
	Lack of the Actor	The reader should know which step is performed by which actor. Thus, every step in a use case should be an action that is performed by one particular actor.
	Misusing Tenses and Verb Forms	Use cases should be written in a way which is highly readable for everyone. Therefore the action ought to be described from the user point of view. In order to ensure this approach, the present simple tense and active form of a verb should be used.
	Using Technical Jargon	Technical details should be kept outside of the functional requirements specification.
	Conditional Steps	Conditional sentences (*if condition then action)* is preferred by computer scientists, but it can confuse the customer. Especially it can be difficult to read when nested *if* statement is used in a use case step. Use cases should be as readable as possible. Such a style of writing makes it complex, hard to understand and follow.

Table 62 - Use Case Checklist of Text2Test (Sinha et al., 2010)

Condition of interest	Description
Stylistic checks	For English sentences e.g., voice use of actions of recognized kinds, use of anaphora.
Complexity checks	For the number of actions in a statement, the number of statements in a use case, and so on.
Completeness checks	Of use case statements e.g., missing actors and actions, missing parameters.
Structural checks	For the model e.g., consistent use of aliases, dangling use case references.
Flow checks	For data and control flow e.g., attempts to *use* items before they are *created*.
Concurrency-related checks	e.g., for possibly concurrent actions or possibly non-serializable behaviors.
Inter-model checks	To compare the actors and items referenced in a use case to an associated domain model.

Table 63 - Common use case defects (Liu et al., 2014)

Defect	Description
Inconsistent step numbering defect	Inconsistent step numbering captures the situation where the sentence numbers of main flow or alternative flow are not consistent. This may lead to incorrect step referencing.
Use case contains the unclear alternative flow starting step defect	In some use cases, the starting step (in main flows) of the alternative flow is not clearly specified. This may lead to ambiguity when merging the alternative flows with the main flow.
Conflict is a function deciding whether two predicates conflict	An overly-strong precondition is one such that inconsistencies between the precondition and the guard conditions of an edge may occur.
The use case contains the missing alternative flow defect.	Missing of alternative flows is the case when the main flow defines some action under some specific condition, however not all the other possible conditions are addressed.
Missing Scenarios	By interacting with the users, the approach must be able to find missing scenarios which are not captured by the use case documents.
Missing Pre/Post-conditions	In most of the use cases, the authors of the use case specifications tend to focus on documenting the action steps and ignore the pre-conditions and post-conditions. Consequently the use cases usually have their preconditions and post-conditions partially documented; missing or redundant conditions also appear frequently. Therefore, it is extremely helpful to provide a way for the users to correct/complete the pre-conditions/post-conditions in order to improve the integrity of the use case document.

A.2.2. Dynamic Analysis of Software Requirements Specification

Table 64 – Consistency and Completeness in CMPN (Lee et al.,1998)

Property	Description
Deadlock	If there exists a set of transitions that are never enabled. This type of flaw is analogous to unreachable code in programs. Since use cases are expected to reflect genuine needs, it is reasonable to require that CMPNs do not contain transitions that are never enabled (**inconsistency**).
Non-determinism	If the reachability analysis reveals the presence of non-deterministic execution paths, the CMPN may be **incomplete** because users may have forgotten to fully specify the constraints associated with the use cases. It must be emphasized that nondeterministic execution paths may have been introduced on purpose and that the final decision can be made only by the domain experts.
Missing toggle place reference	State variables are modeled as toggle places, i.e, a toggle place consists of a pair of places where a place is the negation of the other. When one of these places is missing, the CMPNs are surely **incomplete**.
Toggle place values never modified	State variables are never changed during system operation. CMPNs must contain transitions that are capable of removing or depositing a token from or to the toggle places, respectively. Otherwise, the CMPNs are surely **incomplete**.
Slices with no shared transitions	Slices are likely to contain shared transitions which serve as synchronization points among concurrently executing CMPN slices. The presence of a slice that never interacts with the rest of the system is likely, although not conclusively, to be **incorrect**.

Table 65 – Faults Detected by Time Petri-Nets (Lee et al., 2001)

Fault	Description
Missing information	The name of places or transitions in Petri-Net are not specified; Place, that is not an initial place does not have input arcs; Place, that is not an final place does not have output arcs;
Wrong information	Timing constraints inconsistently - intra-scenario: contradictory timing constraints in a scenario; Timing constraints inconsistently - inter-scenario: specifying two different values on the same interval of two events may cause this inconsistency; Non-determinism: when there is the same event sent from the same object with the same timing constraints but with different objects to receive this event; Incorrectly specified timing constraints.

Table 66 – Use Case Defect Classification (Denger et al., 2005)

Defect Class	Description	Example
Incorrectness	The UC does not match the expected or intended behavior; that is, the information presents in the UC is wrong and does not represent the user requirements.	The flow of a UC does not represent the flow of activities expected by the user.
Incompleteness	The UC does not contain all necessary scenarios. The UC set does not contain all necessary Use Cases. Information that is required for subsequent activities is not present.	An important exception is not specified, a certain actor is not considered.
Inconsistency	A piece of information of a single Use Case or of different Use Cases is described in at least two different, incompatible ways so that there is a contradiction between them.	The quality constraints of a Use Case contradict the event flow. One user action in two different Use Cases requires contradictory system behavior.
Ambiguity	Elements of the Use Case can be interpreted in two or more ways. Thus, it is not clear which of the interpretations are true.	A condition containing "and" and "or" does not explicitly state the required bracketing.

Table 67 – Properties of UC-LTSs (Sinnig et al., 2009)

Property	Description
Well-formedness	All use case steps and extensions IDs be unique. For every step or extension reference, there exists a corresponding use case step or use case extension within the same use case, respectively. There are not circular inclusions in include relationships. The last element of every use case step sequence be either Goto, Success or Failure.
Livelock	Phenomenon, where an application performs an infinite sequence of internal actions. Erroneous use case specifications may contain loops of internal system steps.
Refinement	Verify whether a refining use case model has the same traces and exposes the same non-determinism (existing nondeterministic transitions are preserved).

Table 68 - Properties of Timed and Controlled Petri-Nets (Zhao and Duan, 2009)

Property	Description
Completeness	All places and transitions are specified by particular names; Not isolated subnet exists in the TCPN model of each use case;
Consistency	The TCPN model itself is consistency, i.e. the TCPN is live; TCPN models of related use cases are consistency, i.e. TCPN model of the use case U is consistent with that of U's include use case; TCPN model of the use case U is consistent with that of U's base use case.
Correctness	The reachability graph of TCPN model is correct; The TCPN models is bounded; The time delay of the transition of TCPN models is valid.

Table 69 – Properties of Reactive Petri-Nets (Somé, 2010)

Property	Description
Balancedness	The absence of connection between parallel place/transitions.
1-safety	If for all reachable marking (state) M, each place in Petri-Net contains at most one token. These two properties are sufficient conditions for the generation of structure-preserving State- Charts from Petri-Nets
Lack of non-determinism	The Petri-Nets obtained from use cases are devoid of non-determinism.