Lex2Test: A Method for Generating Test Cases Based on the Language Extended Lexicon and the Requirements Specification

Julieta Screpnik[1], María Alejandra Paz Menvielle[2], Leandro Antonelli[3]

[1]Fac. de Informática, UNLP, La Plata, Bs As, Argentina
[2]Fac. Regional Córdoba,UTN, Córdoba, Argentina
[3]Lifia, Fac. de Informática, UNLP, La Plata, Bs As, Argentina
[3]CAETI, Fac. de Tecnología Informática,UAI, Bs As, Argentina

julietascrepnik@info.unlp.edu.ar
mpaz@frc.utn.edu.ar
lanto@lifia.info.unlp.edu.ar

Abstract. *Software testing plays a fundamental role in the software development lifecycle, because it helps to verify and ensure the quality of the products being built. The results obtained during this stage help to reinforce confidence in the final product while providing valuable insights to support decision-making among project stakeholders. Given the increasing complexity of today's software systems and the growing demands of clients, organizations must adopt innovative strategies to identify defects early and reduce the risks of failure. This article extends the method originally presented in Design of Test Cases Based on the Language Extended Lexicon and the Requirements Specification. In this extended version, the method has been enhanced by incorporating active participation from both the testing team and the client during the test-case generation stage. Additionally, the third step integrates an AI agent acting as an expert tester in functional testing, collaborating with the testing team to refine and improve the generated test cases.*

1 Introduction

Nowadays, the software industry faces the challenge of developing high-quality products within limited timeframes and budgets. Software is now required in almost every domain, and the requested products are becoming more complex as organizations try to stand out in a highly competitive market, where technological innovation has become a key differentiator. The impact of software on our society and culture continues to be profound—not only because of its central role in the digital transformation processes of organizations and governments, but also due to its direct

Cadernos do IME - Série Informática
e-ISSN: 2317-2193 (online)
DOI: 10.12957/cadinf.2026.94933

influence on people's daily lives. It affects how individuals communicate and access information, as well as how they work, learn, and consume services. As its importance continues to grow, the software community persistently seeks to develop technologies that make the construction of high-quality computer programs simpler, faster, and more cost-effective [Pressman, 2010].

Quality is not a new concept, as it has been the subject of study over time by various experts in the field. The Real Academia Española (RAE) [Real Academia Española] defines it as a property or set of properties inherent to something that allows its value to be judged. According to Jerry Weinberg, "quality is value to some person," a notion later extended by Cem Kaner, who stated that "quality is value to a person who matters" [Toledo, 2020]. According to the IEEE [IEEE, 1990], software quality is "the degree to which a system, component, or process meets specified requirements and customer or user needs or expectations." Similarly, ISO 9000 [ISO, 2015] specifies that quality is the degree to which a set of inherent characteristics of an object (product, service, process, person, organization, system, or resource) fulfills requirements. In parallel, Boehm and his collaborators (1978) identified fifteen important software quality attributes, which are listed in Table 1 [Sommerville, 2011].

Table 1. Attributes of Software Quality.

Protection	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Flexibility	Modularity	Efficiency
Robustness	Complexity	Learnability

Consequently, software quality is not only concerned with whether the system's functionality has been correctly implemented, but also depends on the non-functional attributes of the system. These attributes are related to the software's reliability, usability, efficiency, and maintainability. They are usually applied to the system as a whole, rather than to specific features or individual services of the system [Sommerville, 2011].

Deming [Deming, 1989] argues that the difficulty in defining quality lies in translating users' future needs into measurable characteristics, so that a product can be designed and manufactured to provide satisfaction at a price the customer is willing to pay. However, such needs are highly volatile. For this reason, quality can be seen as a subjective concept, influenced by factors such as age, context, and time. Moreover, clients usually do not know exactly what they want or expect from the product until they have it in their hands, which makes quality a difficult goal to achieve. In addition, the notion of quality is relative even for the same person, whose perspective may

change over time. For these reasons, quality is a concept that is difficult to define and even more complex to measure when determining whether a product meets the expected level of quality.

In this context, the generated test cases are often incomplete and inconsistent, which prevents an exhaustive verification of the system's behavior. At times, it is difficult to understand how testers fail to detect certain obvious errors. In many organizations, testers are not adequately prepared to perform the demanding task of testing increasingly complex software products [Serna and Arango, 2011]. This difficulty generally stems from a lack of deep understanding of the specified requirements and limited knowledge of the application domain. It is important to note that, although numerous studies have been conducted over the years in the field of software testing, the authors have not identified requirement-based methods or techniques that directly address the problem described, nor those that explicitly consider the client's needs and expectations regarding the product.

Working with requirements written in natural language makes this process considerably more complex [Denney and Pai, 2018]. Natural language requirements, on the other hand, often present issues such as ambiguity, lack of coherence, and inaccurate or incomplete data [Carvalho et al., 2013]. The requirements of any software project serve as its foundation; without clear requirements, testing cannot be effectively performed, and high-quality software must comply with its specified requirements [Medeiros et al., 2018]. Moreover, errors in requirements account for approximately 56% of all software problems. The most common reason for project failure is the absence of well-defined requirements, which are often written in plain English or other natural languages. Ambiguity is one of the key inherent characteristics of natural language requirements [Farooq and Tahreem, 2022]. Therefore, the testing phase plays a fundamental role in ensuring that the correct software product is built in accordance with the organization's processes and standards, as well as the client's needs and expectations reflected in the requirements.

This work presents an extension of a method previously described [Screpnik et al., 2025], which proposes an innovative approach to the design of test cases in Gherkin format, using the Language Extended Lexicon and the Software Requirements Specification as inputs. The Language Extended Lexicon (LEL) [Leite and Franco, 1990] is a model that allows representing and documenting, through hypertextual technology, a set of symbols that express the language of the application domain without requiring a full understanding of the problem. It provides a description of the significant terms within the macrosystem, constraining the external language by using symbols defined within the LEL itself, while minimizing the use of external symbols unrelated to the domain language. The Software Requirements Specification (SRS), in turn, is a document that details the fundamental aspects of the software to be developed,

aiming to help all stakeholders reach a shared understanding of the system and to avoid possible misunderstandings [Laplante, 2018]. In this extended version, the method has been enhanced by incorporating the active participation of both the testing team and the client in the test-case generation step. Furthermore, in the third step, an artificial intelligence agent is integrated, assuming the role of an expert functional tester who collaborates with the testing team in refining the generated test cases.

In this sense, the proposed approach gives the client a central role during the entire process, since their knowledge of the system is essential for building high-quality test cases. Their collaboration enables the identification and prioritization of the most relevant and critical scenarios that should be considered during the testing stage, thereby contributing significantly to the final quality of the product. In addition, the incorporation of an artificial intelligence agent in the refinement step enriches the process with an additional layer of expert analysis. By assuming the role of a tester specialized in functional testing, this agent facilitates the detection of inconsistencies, redundancies, or omissions that might otherwise go unnoticed by human testers. In this way, the precision and completeness of the generated test cases are improved.

The structure of this article is as follows: Section 2 introduces the fundamental concepts necessary to understand the proposed method; Section 3 describes the proposal in detail; Section 4 presents the related work; and Section 5 provides the conclusions along with directions for future research.

2 Background

This section introduces the fundamental concepts that provide the theoretical basis for understanding the proposed method.

2.1 Language Extended Lexicon

The Language Extended Lexicon (LEL) [Leite and Franco, 1990] is a representation of the symbols used in the language of the application domain, aimed at capturing the vocabulary of a given system. Its main purpose is to enable software engineers to understand the language used by domain experts, including the terms and expressions they employ. The symbols are generally the words or phrases that users most frequently use, as well as other words or expressions that, regardless of frequency, are relevant to the problem domain. A list is then created containing all the identified symbols. The semantics of each symbol are represented through one or more notions and one or more impacts. The notion explains what the symbol is, while the impact describes how it affects the system. Therefore, each symbol has a name that identifies it, along with a notion and an impact that describe it [Hadad et al., 1996]. Finally, each LEL symbol belongs to one of the categories detailed in Table 2.

Table 2. Categories of the LEL [Antonelli et al., 2014].

Category	Characteristics	Notion	Behavioral Responses
Subject	Active elements which perform actions	Characteristics or condition that subject satisfies	Actions that subject performs
Object	Passive elements on which subjects perform actions	Characteristics or attributes that object has	Actions that are performed on object
Verb	Actions that subjects perform on objects	Goal that verb pursues	Steps needed to complete the action
State	Situations which subjects and objects can be in	Situation represented	Actions that must be performed to change into another state

2.2 Software Requirements Specification

The IEEE 29148-2018 standard [ISO/IEC/IEEE, 2018] defines the Software Requirements Specification (SRS) as the specification for a particular software product, program, or set of programs that perform certain functions in a specific environment. The SRS indicates the precedence and criticality of the requirements. It defines all the required capabilities of the specified software product to which it applies, and also documents the conditions and constraints under which the software must operate, as well as the intended verification approaches for the requirements. Typically, the SRS is prepared by the requirements engineer during the analysis phase, in which the client plays a fundamental role by expressing their expectations and needs, thereby contributing to the definition of the scope of the product to be developed.

2.3 Gherkin Template for Specifying Test Cases

Behavior-Driven Development (BDD) is an agile software development approach whose primary purpose is to foster communication and collaboration among all stakeholders involved in the product — including developers, analysts, testers, and clients — through the use of a shared language. This approach employs several languages, among which Gherkin is the most widely used. Gherkin is a semi-natural

language with minimal structure, commonly used in the software development industry. It allows clients to describe the desired system behavior in a way that developers can easily understand [Cauchi et al., 2016]. A Gherkin specification consists of a set of features, each composed of scenarios written as statements (known as steps) that begin with the keywords Given, When, Then, and And. Figure 1 presents an example of a test scenario corresponding to the registration of a banking package.

```
Feature: Generate Package Registration
Scenario: Successful Package Registration
   Given the client exists in the bank's database
   And the client has at least one active savings account
   And the client has an available offer for package registration and an associated credit card
   When the branch analyst accesses the menu option "New Request"
   And selects an available package from the list according to the current offer
   Then the system displays a confirmation message "Package request successfully created."
```

Fig. 1. Example of a test scenario following the Gherkin template.

The Given keyword is used to describe a precondition, that is, the context in which the scenario is relevant. The When statement describes the action that can be performed once the precondition is met, while the Then statement specifies the postcondition that should hold after the action is executed. Finally, the And keyword is used as a linking mechanism when the author of a scenario needs to divide a precondition, action, or postcondition into several steps [Cauchi et al., 2016].

3 Approach

This section presents an overview of the proposed method and subsequently provides a detailed description of each step.

3.1 Method Overview

This article proposes a sequential method for generating test cases in Gherkin format, based on the symbols defined in the Language Extended Lexicon and the system's functional requirements expressed in natural language within the Software Requirements Specification. As a result, a set of test cases written in natural language and structured in Gherkin format is obtained. The method is organized into the following steps: application of suggestions to improve the LEL, generation of test cases, and application of suggestions to improve the test cases. Figure 2 provides a summarized illustration of the proposed method.

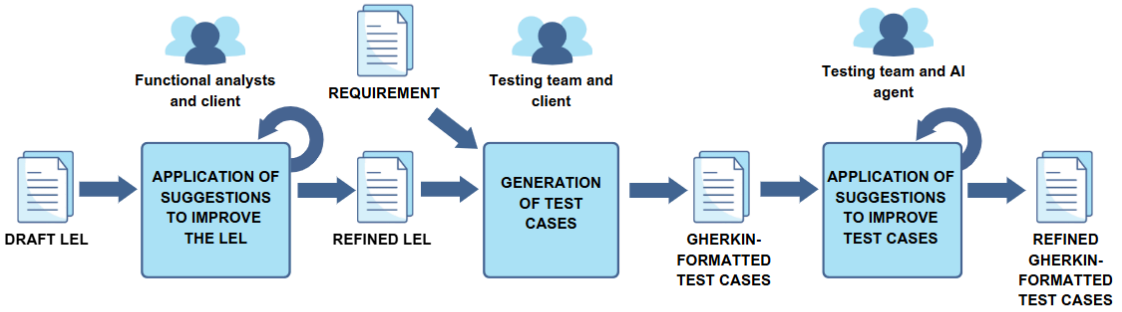

Fig. 2. Proposed Method.

The main contribution of this approach is the syntactic and semantic correspondence established between the requirements described in the Software Requirements Specification and the symbols detailed in the Language Extended Lexicon. This relationship facilitates the construction of more accurate and representative test scenarios within the application domain. In addition, the selected symbols from the LEL help identify the critical elements that require validation, ensuring that the generated test scenarios accurately reflect the domain of the application. Finally, the information provided in the notion and impact fields of the extracted LEL symbols adds a higher level of precision and granularity to the design of the test cases, contributing to broader test coverage.

3.2 Application of suggestions to improve the LEL

The first step of the method consists of refining the symbols identified in the Language Extended Lexicon to obtain a more complete, meaningful, and detailed representation of the domain language, which serves as the foundation for the generation of test cases. At this stage, functional analysts and the client participate, as they possess the necessary domain knowledge. The refined LEL produced as the output of this step is intended to capture both the business domain knowledge and the knowledge of the application to be developed. This step follows an iterative approach, allowing analysts and the client to progressively refine the LEL symbols through successive cycles until the desired level of accuracy is achieved. Tables 3-6 present a set of suggestions for improving the LEL Symbols.

Table 3. Suggestions for Improving the Subject Symbol.

Category	Notion	Behavioral Responses
Subject	What is the subject's objective? Why does it act? What essential characteristics should it have? Which characteristics should it not have? Can the subject assume different roles or hierarchies within the system? Are subjects classified by priority? What knowledge, skills, or information should the subject possess to interact with the system? Does the subject depend on other actors to fulfill its function?	Which actions generate critical effects in the system? How is the subject expected to interact with the system? Are there actions performed by the subject that change the state of other elements within the system? What changes should the subject's actions produce on the system? What external events may modify its behavior within the system? Are there any restrictions on the actions it can perform?

Table 4. Suggestions for Improving the Verb Symbol.

Category	Notion	Behavioral Responses
Verb	How do we want to perform this action? (Considering non-functional requirements) Are there similar or related actions — as variants of the same action? Are there exceptions or limitations? Are there operations that are critical or main within the system? What preconditions must be satisfied to perform the action? Can the action be executed in parallel with others? How long should the action take to complete?	What criteria must be met to consider that the action has been successfully completed? Under what circumstances could the action be considered to have failed? What impact does the execution of a given action have on the states of the affected elements? What should happen if the action is unexpectedly interrupted? What business rules constrain or condition the execution of the action?

Table 5. Suggestions for Improving the Object Symbol.

Category	Notion	Behavioral Responses
Object	What essential characteristics should it have? Which characteristics should it not have? Does it have any classification or categorization? Are there dependencies among the objects? Are objects classified by priority? Which attributes of the object are mandatory and which are optional?	What consequences do the subject's actions have on the object? How does the presence or absence of the object affect the system's flow? What changes should the object undergo after each interaction with the subject? How should the system react if the object is in an incorrect or incomplete state? What relationships does it maintain with other objects in the system?

Table 6. Suggestions for Improving the State Symbol.

Category	Notion	Behavioral Responses
State	Can there be substates within the system? Can there be states that depend on other states? Are there situations that trigger a state change in multiple system components? Do intermediate states exist? Can the same state be reached through different paths or transitions? What constraints exist for returning to a previous state?	How is it validated that the system has reached a specific state? What effect does the transition to a specific state have on the system? What conditions are necessary for a state change to occur? What are the consequences if the system remains in a given state for an extended period of time? What are the consequences if the system fails to transition to a specific state? What internal or external events trigger a state change?

As a main recommendation, it is suggested to write complete statements for each LEL symbol, explicitly incorporating the subjects and objects involved with the highest possible level of specificity. This approach allows for the extraction of more detailed information and facilitates the generation of more consistent test cases that are better aligned with the system's needs.

3.3 Generation of test cases

This step takes as input the Software Requirements Specification and the refined Language Extended Lexicon obtained in the previous step. Both the testing team and the client participate in this stage: the testers, due to their technical expertise in the testing domain, and the client, for providing knowledge about the critical and priority scenarios that require validation. The process consists of a set of substeps that enable the identification of LEL symbols requiring special attention, which will later serve as the foundation for the creation of test cases. These substeps are as follows:

Requirement Selection: the IEEE 29148-2018 standard [ISO/IEC/IEEE, 2018] defines a requirement as a statement that translates or expresses a need, along with its associated constraints and conditions. A requirement may be written in natural language or in another formal language. When expressed in natural language, the statement should include a subject and a verb, along with any additional elements necessary to adequately convey the informational content of the requirement. Thus, a functional requirement is selected from the Software Requirements Specification, which serves as the basis for the following substeps.

LEL Symbol Identification: it's essential to understand the relationship established between the LEL symbols and the requirements. The main verbs present in the requirements are usually associated with the names of verb symbols in the LEL, as they express actions performed by a subject interacting with the system. Similarly, the nouns appearing in the requirements are generally linked to object and subject symbols in the LEL, reflecting elements that interact with or are used within the system. In this context, subjects represent the actors who use the system, while objects correspond to internal components of the system. Likewise, the conditions included in the requirements can be incorporated into the notion of object symbols in the LEL, since these conditions frequently refer to specific attributes of system components or entities. On the other hand, some nouns found in the requirements contribute to enriching the notion of subject symbols by referring to inherent characteristics of the actors involved. Consequently, the LEL symbols are analyzed and selected to identify those whose terms match the previously selected requirement, establishing a syntactic and semantic correspondence between both artifacts.

Extraction of Key Attributes: the terms (symbols) are defined through two attributes: notion and impact. The notion describes the intrinsic and substantial characteristics of the symbol (denotation), while the impact (connotation) describes the relationship between the term being defined and other related terms [Antonelli et al., 2012]. As a result, the notion and impact columns of the symbols selected in the previous step are extracted in order to achieve a more comprehensive and precise coverage in the creation of test cases.

Classification into Test Elements: there is a close relationship between the test cases and the LEL. The notion of the subjects and objects involved in the impact of the verb symbol in the LEL can be incorporated into the preconditions of a test case. Likewise, the steps described in the impact of that symbol, along with the related subjects and objects, are reflected in the execution steps of the test case. The notion of the verb symbol in the LEL can be associated with the expected result of a test case, while its name, along with the corresponding subject and object, can be used in the test case title. Furthermore, the requirement itself has a significant relationship with the test case: the nouns and conditions expressed in the requirement may be incorporated into the preconditions, while the main verb relates to both the title and the expected result of the test case. For these reasons, the LEL symbols selected in the previous substeps are classified into the categories of preconditions, execution steps, and expected results, considering both positive and negative scenarios.

Creation of Test Cases in Gherkin Format: the test cases are written in Gherkin format, using the keywords Given, When, and Then to clearly specify the preconditions, execution steps, and expected results of both positive and negative scenarios. For their construction, the previously classified LEL symbols are considered together with the selected requirement.

3.4 Application of Suggestions for Improving Test Cases

As the final step, it is recommended to refine the previously generated test cases so that they more accurately reflect the main or critical scenarios defined in the previous stage. In this process, an artificial intelligence (AI) agent participates, configured to act as an expert functional tester. The agent analyzes the test cases written in Gherkin format and proposes adjustments to improve their completeness. However, the final decision regarding the acceptance or incorporation of these suggestions lies with the testing team, who are responsible for verifying the validity and consistency of the test cases. The process follows an iterative approach, in which the AI agent generates refinement proposals based on a predefined set of analytical questions, and the testing team evaluates these proposals. This cycle can be repeated as many times as necessary, ensuring that the test cases evolve until they reach the required level of detail. To guide this improvement process, the following questions are proposed:

- Is it a new functionality or a requested change to an existing one?
- If it is a requested change to an existing functionality, does it imply a functional or technical modification?
- If it is a new functionality, how does this new feature impact the current system?
- What is the scope and what are the limitations of the functionality to be tested?
- Which users or roles interact with it, and what permissions are required?

- Are there primary or critical user flows that must be considered? What happens if an unauthorized user attempts to execute the flow?
- Are there secondary user flows that must be considered?
- What happens if multiple users attempt to execute this functionality simultaneously?
- Are there environment-specific differences (dev/qa/uat/prod) that affect the test cases?
- What is the data input? What formats, ranges, and constraints apply to each field?
- How does it behave with missing, null, duplicate, or corrupted data?
- What is the expected output? Does it generate any output in databases, files, or logs?
- What logs, metrics, and traces are required to diagnose failures?
- What is the defined rollback strategy, and what contingency mechanisms are in place in case of failure?
- What web services and APIs does the functionality consume? What happens if an API or service becomes slow, fails, or returns unexpected error codes?
- How is version compatibility between services handled?
- Which databases and tables are queried? What operations does the functionality perform on the database (read, write, update, delete)?
- What happens if the database is unavailable or returns an error? How are duplicate or inconsistent records managed?
- Is the functionality related to other products or systems within the organization (in terms of dependencies or integrations)? What happens if a dependent system does not respond or responds with a delay?

It's important to note that, although these questions do not explicitly indicate which modifications should be made to the structure of the test cases, their purpose is to serve as analytical triggers to identify potential adjustments that may be necessary. For example, a new functionality might lead to the creation of new scenarios, whereas a change to an existing functionality could involve modifications to the preconditions, execution steps, or expected results of the generated test cases.

4 Evaluation

The validation of the proposed approach was carried out through an application case focused on the second and third steps of the method. The second step corresponds to the design of test cases in Gherkin format, which uses as input the previously refined Language Extended Lexicon and the Software Requirements Specification. The third step focuses on the refinement of the previously generated test cases, performed collaboratively by an AI agent and the testing team.

The evaluation involved the participation of five members from a software development project in the banking sector in Argentina, who had previously participated in the first validation of the method. The project involved developing and maintaining a system for managing financial transactions, customer account administration, and product operations, while meeting regulatory standards and high levels of security. All participants had experience in the field of software development and performed the role of quality assurance analysts, with an average of three years of experience in the area. It is worth noting that none of them had prior experience using the Language Extended Lexicon. Therefore, they received training on the proposed method before the experiment. During the validation process, all participants collaborated in the generation of test cases. Their task consisted of selecting a requirement from the Software Requirements Specification and, based on it, identifying the LEL symbols with syntactic and semantic correspondence to the selected requirement. From the identified symbols, the attributes Notion and Impact were extracted and then classified into the categories of preconditions, execution steps, and expected results, considering both positive and negative scenarios. Finally, the test cases were written in Gherkin format, using the keywords Given, When, and Then. Subsequently, an AI agent configured to act as an expert functional tester was used to refine the previously generated test cases. The agent was provided with the set of analytical questions defined in the third step, along with the generated test cases, and was instructed to analyze them and propose improvements to make them more complete. Finally, the participants were asked to evaluate the improvement proposals generated by the AI agent, determining whether they were correct and whether they effectively contributed to enhancing the completeness and precision of the test cases.

For this stage, a 7-point Likert scale was used to measure the participants' perceptions regarding the ease of use, usefulness, and intention to use of the proposed method. Likert scale was devised in order to measure 'attitude' in a scientifically accepted and validated manner in 1932 [Edmondson, 2005; McLeod]. The original Likert scale is a set of statements (items) offered for a real or hypothetical situation under study. Participants are asked to show their level of agreement (from strongly disagree to strongly agree) with the given statement (items) on a metric scale. Here all the statements in combination reveal the specific dimension of the attitude towards the issue, hence, necessarily inter-linked with each other [Singh, 2006]. The 7 point scale provides more varieties of options which in turn increase the probability of meeting the objective reality of people. As a 7-point scale reveals more description about the motif and thus appeals practically to the "faculty of reason" of the participants [Chang, 1994; Cox, 1980].

The evaluation results showed that participants tended to select "agree" when rating the ease of use, usefulness and intention to use of the proposed method,

suggesting a positive perception of its practicality and applicability. Based on these results, it can be inferred that the method could be applied by professionals in real software projects, even by those without previous experience using the LEL.

5 Related works

Over time, several scientific papers have been published with the aim of defining a process for designing test cases from requirements. However, the authors have not identified approaches that consider the joint use of the Language Extended Lexicon and the Software Requirements Specification as input artifacts. In this context, a systematic literature review was conducted to examine methods based on the semantic and syntactic analysis of requirements, with the goal of contextualizing and comparing existing approaches with the proposed method, particularly in relation to the "Generation of test cases" step.

Dwarakanath and Sengupta [Dwarakanath and Sengupta, 2012] present a tool that analyzes the sentences of a functional requirements document using a syntactic parser to determine their testability. If a sentence is complex, it is divided into simpler ones, from which test intentions are generated to represent the aspects to be evaluated. These intentions are then grouped and sequenced to generate one or more positive and negative test cases. Similarly, Gutiérrez Rodríguez et al. [Gutiérrez Rodríguez et al., 2011] propose a process that converts the concrete syntax of a functional requirement into an abstract syntax by developing metamodels and defining QVT (Query-View Transformation) transformations, with the goal of obtaining test scenarios and test values that can later be used to generate functional test cases. Along the same lines, Alcaraz [Alcaraz, 2021] developed a context-free grammar that enables both syntactic and semantic analysis of requirements, from which test cases are automatically generated. The author notes that to apply this approach to other types of requirements, the grammar must be extended, as it was specifically designed for the analysis of hardware functional requirements.

Verma and Beg [Verma and Beg, 2013], in contrast, propose an approach for generating test cases from requirements expressed in natural language using natural language processing (NLP) techniques. To achieve this, a preprocessing phase is performed to normalize the text. Then, part-of-speech (POS) tagging is applied to assign a grammatical category to each word in a sentence. Then, a syntactic parser generates trees for each requirement, and graphs are used to represent the knowledge contained in the requirements. Finally, the graph is traversed using the boundary value technique to generate test cases. Analogously, Sneed [Sneed, 2007] presents an approach for generating test cases from requirements written in natural language. A text analyzer is used to examine both functional and non-functional requirements, identifying relevant

objects, states, and actions by analyzing nouns and their context, with the goal of generating test cases from the specified events. In a similar way, Ansari et al. [Ansari et al., 2017] propose a method that automatically generates test cases from functional requirements expressed in natural language, also using NLP techniques. The system identifies conditional structures of the form “if...then” within the requirements and extracts from them the test steps, input data, and expected results, organizing them into a structured table.

On the other hand, some approaches propose the use of various artifacts, usually created during the analysis phase, as inputs to the method for test case generation. Freudenstein et al. [Freudenstein et al., 2018] also employ a tool to model functional requirements using cause–effect graphs, from which test cases are automatically designed. In the same way, Ibrahim et al. [Ibrahim et al., 2007] developed a tool that generates use case diagrams, event flows, and sequence diagrams from functional requirements, and uses these artifacts both for the creation and validation of the generated test cases. Chatterjee and Johari [Chatterjee and Johari, 2010], in contrast, developed the STATEST 1.0.0 tool, which formalizes requirements by transforming use cases and their scenarios into State Transition Diagrams (STDs), which are then used to generate test cases. At the same time, Lafi et al. [Lafi et al., 2021] propose a method for generating test cases from use case descriptions. For this purpose, the textual description of the use case and its UML diagram are used to extract the necessary information. A control flow graph and a natural language processing table are then created from the extracted information. Using these elements, test paths are generated, which are later used to create the test cases.

Although existing approaches have achieved significant progress, none of them effectively address the problems arising from the ambiguity, incompleteness, and inconsistency of requirements. In this regard, our proposal incorporates the use of the Language Extended Lexicon together with the Software Requirements Specification as input to the test case generation method, as an alternative solution. The purpose is to increase the completeness and accuracy of the generated test cases, while promoting broader coverage and reducing the ambiguities inherent in natural language requirements that can affect the quality of the final software product.

6 Conclusions and future work

The article proposes a sequential method for generating test cases in Gherkin format, using the Software Requirements Specification and the Language Extended Lexicon as input. Although in this paper the method is presented using an SRS as input, it can also be applied using user stories or use cases for test case creation. The main challenge associated with the proposed approach lies in the fact that the effectiveness of the

generated test cases depends on the clarity, completeness, and accuracy of the requirements defined in the SRS, as well as on the proper structure and definition of the LEL symbols. The presence of ambiguous or inconsistent requirements, or a poorly structured LEL, may result in inaccurate or inadequate test cases, compromising the effectiveness of the testing process.

Future work will focus on making the method fully iterative, so that both the LEL and the refined Gherkin-formatted test cases obtained in previous iterations can be reused as input in subsequent cycles. Furthermore, it is suggested that the "Generation of test cases" step adopt an iterative approach, allowing for progressive adjustments to the test cases according to the growth in size and complexity of the software under development. An additional line of work involves incorporating artificial intelligence into the "Generation of test cases" step, applying natural language processing techniques to perform a syntactic and semantic analysis of the requirements and LEL symbols, with the goal of producing more precise and representative test scenarios. Finally, a more comprehensive evaluation is planned, involving professionals with different roles and levels of experience, who will design both positive and negative scenarios with the purpose of analyzing the robustness, feasibility, and applicability of the proposed method.

References

Alcaraz, A. (2021). *Generación automática de casos de prueba a partir de requerimientos funcionales utilizando gramáticas libres de contexto*. Dirección de Posgrado, CIATEQ A. C. Centro de Tecnología Avanzada.

Ansari, A., Baig, M. S., Fatima, A. S. and Shaikh, T. (2017). Constructing test cases using natural language processing. Proceedings of the 3rd International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB17), IEEE, India.

Antonelli, L., Rossi, G., Leite, J. C. S. P. and Oliveros, A. (2012). Deriving requirements specifications from the application domain language captured by Language Extended Lexicon. Workshop on Requirements Engineering (WER), Buenos Aires, Argentina, April 24–27.

Antonelli, L., Rossi, G., Leite, J. C. S. P. and Oliveros, A. (2014). Language extended lexicon points: Estimating the size of an application using its language. Proceedings of the IEEE 22nd International Requirements Engineering Conference (RE), 263–272. DOI: 10.1109/RE.2014.6912268.

Carvalho, G., Falcao, D., Barros, F., Sampaio, A., Mota, A., Motta, L. and Blackburn, M. R. (2013). NAT2TEST(SCR): Test case generation from natural language

requirements based on SCR specifications. Proceedings of the 28th Annual ACM Symposium on Applied Computing. DOI: 10.1145/2480362.2480591.

Cauchi, A., Colombo, C., Francalanza, A., Micallef, M. and Pace, G. (2016). Using Gherkin to extract tests and monitors for safer medical device interaction design. Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS ’16), 275–280.

Chang, L. (1994). A psychometric evaluation of 4-point and 6-point Likert-type scale in relation to reliability and validity. Applied Psychological Measurement, 18(3), 205–215.

Chatterjee, R. and Johari, K. (2010). A prolific approach for automated generation of test cases from informal requirements. SIGSOFT Software Engineering Notes, 35, 1–11. DOI: 10.1145/1838687.1838702.

Cox, E. P. (1980). The optimal number of response alternatives for a scale: A review. Journal of Marketing Research, 17(4), 407–422.

Deming, W. E. (1989). Calidad, productividad y competitividad: La salida de la crisis. Díaz de Santos.

Denney, E. and Pai, G. (2018). Tool support for assurance case development. Automated Software Engineering, 25(3), 435–499.

Dwarakanath, A. and Sengupta, S. (2012). Litmus: Generation of test cases from functional requirements in natural language. Lecture Notes in Computer Science, 58–69. DOI: 10.1007/978-3-642-31178-9_6.

Edmondson, D. R. (2005). Likert scales: A history. Proceedings of the 12th Conference on Historical Analysis and Research in Marketing (CHARM), 1–8. California, USA.

Farooq, M. S. and Tahreem, T. (2022). Requirement-based automated test case generation: Systematic literature review. VFAST Transactions on Software Engineering. DOI: 10.21015/vtse.v10i2.940.

Freudenstein, D., Radduenz, J., Junker, M., Eder, S. and Hauptmann, B. (2018). Automated test-design from requirements: The Specmate tool. Proceedings of the 5th International Workshop on Requirements Engineering and Testing (RET ’18). DOI: 10.1145/3195538.3195543.

Gutiérrez Rodríguez, J. J., Escalona Cuaresma, M. J. and Mejías Risoto, M. (2011). Generación de pruebas del sistema a partir de la especificación funcional. Tesis doctoral, Universidad de Sevilla, España.

Hadad, G., Kaplan, G., Oliveros, A. and Leite, J. C. S. P. (1996). Integración de escenarios con el Léxico Extendido del Lenguaje en la elicitación de requerimientos: aplicación a un caso real. Departamento de Investigación, Universidad de Belgrano y Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

Ibrahim, R., Saringat, M. Z., Ibrahim, N. and Ismail, N. (2007). An automatic tool for generating test cases from the system's requirements. Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT 2007). DOI: 10.1109/cit.2007.116.

ISO (2015). ISO 9000:2015 – Sistemas de gestión de la calidad – Fundamentos y vocabulario. Organización Internacional de Normalización. Disponible en: https://www.iso.org/obp/ui/es/#iso:std:iso:9000:ed-4:v1:es

ISO/IEC/IEEE (2018). International Standard – Systems and Software Engineering — Life Cycle Processes — Requirements Engineering. ISO/IEC/IEEE Std 29148-2018 (2nd ed.).

Lafi, M., Alrawashed, T. and Hammad, A. M. (2021). Automated test cases generation from requirements specification. Proceedings of the 2021 International Conference on Information Technology (ICIT). DOI: 10.1109/ICIT52682.2021.9491761.

Laplante, P. A. (2018). Requirements Engineering for Software and Systems. Taylor & Francis Group.

Leite, J. C. S. P. and Franco, A. P. M. (1990). O uso de hipertexto na elicitaçao de linguagens da aplicaçao. Brazilian Symposium on Software Engineering. DOI: 10.5753/sbes.1990.24172.

McLeod, S. (2014). Likert scale. Simply Psychology. Disponible en: http://www.simplypsychology.org/Likert-scale.html/pdf (consultado el 16 de octubre de 2025).

Medeiros, J., Vasconcelos, A., Silva, C. and Goulão, M. (2018). Quality of software requirements specification in agile projects: A cross-case analysis of six companies. Journal of Systems and Software, 142, 171–194. DOI: 10.1016/j.jss.2018.04.064.

Pressman, R. S. (2010). Ingeniería del software: un enfoque práctico. McGraw-Hill.

Real Academia Española (s.f.). Calidad. Diccionario de la lengua española (23.ª ed.). Disponible en: https://dle.rae.es/calidad

Serna, M. E. and Arango, F. (2011). Software testing: More than a stage in the life cycle. Revista de Ingeniería, Universidad de los Andes, 35, 34–40. ISSN 0121-4993.

Singh, Y. K. (2006). Fundamentals of Research Methodology and Statistics. New Age International (P) Ltd. Publishers, New Delhi.

Sneed, H. M. (2007). Testing against natural language requirements. Proceedings of the 7th International Conference on Quality Software (QSIC 2007), Portland, OR, USA, 380–387. DOI: 10.1109/QSIC.2007.4385524.

Sommerville, I. (2011). Ingeniería del software. Addison-Wesley.

Screpnik, J., Menvielle, M. A. P. and Antonelli, L. (2025). Diseño de casos de pruebas a partir del Léxico Extendido del Lenguaje y la Especificación de Requerimientos. Proceedings of the 28th Workshop on Requirements Engineering (WER2025), Brazil. DOI: 10.29327/1588952.28-7.

Toledo, F. (2020). Introducción a las pruebas de sistemas de información. Abstracta.

Verma, R. P. and Beg, M. R. (2013). Generation of test cases from software requirements using natural language processing. Proceedings of the 6th International Conference on Emerging Trends in Engineering and Technology (ICETET). DOI: 10.1109/icetet.2013.45.