Delta University Scientific Journal
Journal home page: https://dusj.journals.ekb.eg

Employing NLP Approach for Formulation of Acceptance Tests based on Extracting Conditional Expressions from Requirements

Mostafa M. Ragab, Amany M. Sarhan
Department of Computer and Control Engineering, Faculty of Engineering, Tanta University, Tanta, Egypt

Correspondence: amany_sarhan@f-eng.tanta.edu.eg, Mostafa.m.ragab@f-eng.tanta.edu.eg

ABSTRACT
In the continually changing industry of software engineering, assuring quality and dependability is critical. As systems evolve, requirements analysis and test case production get more complicated, making high coverage difficult to achieve. This paper describes a current way to automatically create acceptance tests that uses Natural Language Processing (NLP) to extract conditional statements from textual requirements. These conditionals are the foundation of test scenarios, and automating their extraction considerably saves the time and mistakes involved with human test case generation. CiRA (Conditionals in Requirements Artifacts), a tool-supported technique, tackles this issue by automatically producing test cases based on conditionals in natural language requirements. CiRA delivers a substantial level of automation in real circumstances through the use of NLP methods. This paper describes a case study with three industry partners—Allianz, Ericsson, and Kostal—in which CiRA successfully created more than 70% of the needed test cases. CiRA also found and developed test cases that were missed during the human test design process, indicating its efficacy in improving the reliability and completeness of acceptance testing. This technique not only speeds up the testing process, but it also provides a greater degree of system quality by including more situations with less manual involvement.

Keywords: *Acceptance testing, Automatic test case creation, Requirements engineering, Natural language processing*

1. Introduction

The fast advancement of software development technology has created an increased demand for efficient and effective software testing techniques. Requirement validation is an essential step in ensuring that the requirements are complete and consistent according to the user needs. Acceptance criteria directly validate the requirements using requirement patterns. By executing tests that align with these patterns, the team can assess whether the software behaves as expected and fulfills the specified requirements. Test execution provides critical feedback on the effectiveness of the requirement patterns and also ensures that the software meets the user expectations, which are often encapsulated in the requirement patterns.

One of the most promising developments in this area is the use of NLP (Natural Language Processing) tools. NLP, a branch of artificial intelligence (AI), is concerned with the interaction of computers and humans using natural language. In the context of software testing, NLP has the ability to automate test case generation and documentation, hence saving time, effort, and money compared to manual testing techniques.

Acceptance tests are used to verify the alignment between end-user requirements and actual system behavior. Each acceptance test comprises a set of test cases that specify specific inputs and expected outcomes. The design of these test cases is a labor-intensive process, accounting for 40-70% of the total testing effort. This complexity arises from two main challenges:

- Challenge 1: Determining the appropriate set of test cases to fully cover a requirement is particularly difficult, especially for complex requirements. A requirement is fully covered if the associated test cases ensure the expected behavior. However, acceptance tests are often created without a systematic approach, leading to incomplete or excessive test cases. Missing test cases can result in undetected system defects, while excessive test cases lead to unnecessary testing efforts and increased maintenance costs. Thus, a balance between comprehensive test coverage and the number of test cases is crucial.
- Challenge 2: Creating acceptance tests remains a predominantly manual task due to insufficient tool support. While existing approaches allow for test case derivation from semi-formal or formal

requirements, they are not well-suited for processing informal natural language (NL) requirements. Studies have shown that NL requirements are common in practice but approaches to derive test cases from them often perform poorly on real-world data due to issues like grammatical errors and limited vocabulary.

This paper depicts the benefits and challenges of using NLP in software testing, focusing on automating test case creation and documentation. We will discuss the key NLP techniques used in this area, real-world applications, and the future of NLP in software testing. This paper aims to develop a tool-supported approach to automatically derive the minimal set of required test cases from NL requirements using Natural Language Processing (NLP). Where Functional requirements often describe system behavior through event relationships, such as "If the system detects an error (e1), an error message shall be shown (e2)." Previous studies indicate that such conditional statements are prevalent in requirements. This paper focuses on these conditionals, using their embedded logic for automatic test case derivation. The paper presents CiRA (Conditionals in Requirements Artifacts), a tool capable of detecting conditional statements in NL requirements, extracting their relationships, and mapping them to a Cause-Effect-Graph for automatic test case derivation.

2. Basics and background

The software requirements specification (SRS) document allows us to codify both functional and nonfunctional needs. The SRS describes the functions and characteristics that the product must fulfill. It also specifies restrictions and assumptions. It is critical to make the SRS clear and understandable for all parties. Use templates that include images to assist arrange and interpret the content. If your requirements are contained in other document formats, include a link to them so that readers may access the information they need.

We begin viewing the requirements as use cases and we can build the use cases diagram as in Fig. 1. Then for each of the use cases, we write down the full specification details as shown in Fig. 2. These descriptions are used to design and implement the system and, in the end, to generate the test cases used to verify the system behaves as the user required.

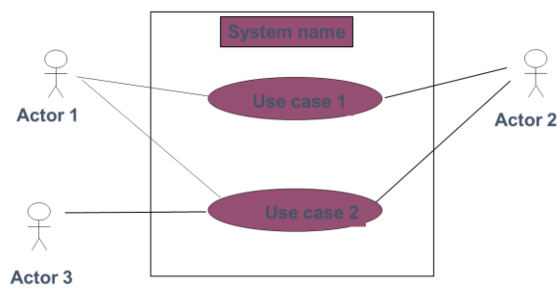

Fig. 1. Use case diagram for a given system with multiusers view

Overview	
Title	[Title of the basic flow use case]
Description	[Short description of the basic flow]
Actors and Interfaces	[Identifies the Actors and Interfaces to components and services that participate in the use case]
Initial status and Preconditions	[A pre-condition (of a use case) is the state of the system that must be present prior to a use case being performed]
Basic Flow	
STEP 1: ... STEP 2: ...	
Post Condition	
[A post-condition (of a use case) is a list of possible states the system can be in immediately after a use case has finished]	
Alternative Flow(s)	
[Alternative flows are described here if needed]	

Fig. 2. Use case description template

Software testing is an essential activity that ensures a specific level of quality in software systems. However, testing requires a significant amount of effort. In its traditional form, human testers (test engineers) do the majority (if not all) parts of software testing manually. One of these steps is test-case design, in which a human tester generates a collection of test cases based on written (formal) requirements, which are frequently stated in natural language (NL). Test-case design is likewise a time-consuming task [1], and practitioners are keen to benefit from any (partially) automated way to extract test suites from requirements [1]. Such a method might save software businesses significant resources that are currently spent manually deriving and documenting test cases from requirements (Garousi et al.).

Many NLP-based techniques have been presented in the literature to decrease the human work of turning natural-language (NL) requirements into test cases. Such a method needs an input set of requirements stated in NL. The textual requirements are then automatically extracted into a collection of test cases using a number of NLP procedures. A test case is one or more inputs (as required) and the expected output(s) (or behavior) for a unit or system being tested. For example, to test an absolute-value function, one would require at least two test cases: one with a positive integer and another test.

With the introduction of large language models (LLMs), the use of natural language processing (NLP) techniques in software testing has become more and more popular. (Kirinuki & Tanno, 2024) examine how LLMs have revolutionized black-box testing, highlighting how they can produce test cases straight from software specifications. They draw attention to the Transformer model's self-attention mechanism, which supports these models' better capacity for generalization. This basic knowledge paves the way for investigating how NLP may help automatically generate acceptance tests by taking conditionals out of requirements.

Alagarsamy et al. offered A3Test, a unique DL-based method to test case creation that includes assertion knowledge and a technique for verifying the coherence of test names and signatures. A3Test seeks to translate domain knowledge from assertion creation to test case generation. A3Test applies domain adaption principles and proposes a verification technique for name consistency and test signatures. They tested its efficiency with 5,278 focused techniques from the Defects4j dataset.

(Kirinuki & Tanno, 2024) argue that the automation of test case generation is a critical area of research, particularly as it pertains to leveraging natural language requirements. The authors categorize existing approaches into those utilizing unified modeling language, formal methods, and natural language analysis techniques. Their investigation into the potential of ChatGPT underscores the model's ability to navigate complex language patterns, making it a suitable candidate for generating high-level test cases that are aligned with specified requirements.

The implications of their findings suggest that by harnessing the capabilities of LLMs, software testing can transition towards a more automated and efficient paradigm. This literature review will further delve into the methodologies and outcomes presented by (Kirinuki & Tanno, 2024), offering a comprehensive analysis of how NLP approaches can enhance the automatic creation of acceptance tests, ultimately contributing to the field's advancement.

Meziane et al. discussed the use of natural language writing using the UML class diagram. The purpose of this work is to reinforce the perspective of generating NL specification from class diagrams by describing several NL based systems. The study demonstrated how to generate semantically sound sentences that explain the structure of UML string names using WordNet.

To create class models, Reynaldo employs controlled NL text of requirements. The report presents some preliminary findings from the text's ambiguity parsing. The report presents the author's research strategy to incorporate requirement validation into the RAVEN project. An automated program called UMGAR was developed by Deva Kumar and colleagues to produce UML analysis and design models from natural language text. To complete this objective, they used Java RAP, Word Net 2.1, and Stanford parser.

Through the development of the SPIDER tool, Sascha et al. presented a round trip engineering process. The study tackled the issues regarding requirements-level defects extending to the design and coding phases. A UML model is provided to the developer by means of the behavioral attributes presented in the NL text. From NL text, Priya More and colleagues have created a UML diagram. A tool known as RAPID has been created by them to analyze the requirements. The RAPID Stemming algorithm, WordNet, and OpenNLP software were utilized to finish the work. The function of ontology in object-oriented software engineering is covered by Waralak et al. The author provides an overview of object modeling and ontology. The development tools and other standards that ontologies can be used with are then covered in the paper by Waralak et al.

Walter and colleagues propose that universal programmability opens up the possibility of programming for every individual. The authors assert that universal programming will be achievable with the integration of NLP, AI, and SE. As a benchmark for NLP needs, the authors are presently working on nlrpBENCH.

2.1 Problem formulation and plan of solution

Where, numerous methods based on Natural Language Processing (NLP) have been presented in the literature to decrease the manual labor involved in turning natural-language (NL) requirements into test cases. An input set of criteria defined in NL is necessary for such an approach. So, the term "NLP-assisted software testing" must be used in this study to refer to all NLP-based methods and instruments that could help with any software testing task, such as the previously covered test-case design and test evaluation.

3. Material and methods

In general, requirements can be categorized as TR, BR, SR, and STR. The collected requirements are divided into informal and non-casual phrases. Following that, the proven casual phrases are labeled and their associations defined. We build strong acceptance test cases based on the requirements for development. For example, a sample requirement of an application is "*When the exit button is pressed, the interface should show a confirmation message to verify the exit and closing status*". This is a casual phrase that connects between the current state and next state with the press of a certain button.

The requirements engineer should be able to inspect and edit the produced model artifacts, even for simple syntactical needs. This pertains to a prior study in which the authors attempted to build executable test cases using a Restricted Test Case Modeling (RTCM) language that limits the manner of creating test cases. This adds an extra burden to the requirement engineers who create formal requirements. Users are expected to review produced OCL limitations before creating test cases. Other research has investigated the idea of creating test cases using Petri Net simulation; however, the interpretability of Colored Petri Nets as presented in the technique may differ depending on the user's degree of competence. Users may struggle to understand intermediate outcomes and may need to fine-tune or adjust predictions to provide credible test cases.

In this work, we alternatively employ NLP for the detection of conditions in the requirement document, then convert them into a graph from which we will build the test cases as shown in fig. 3.

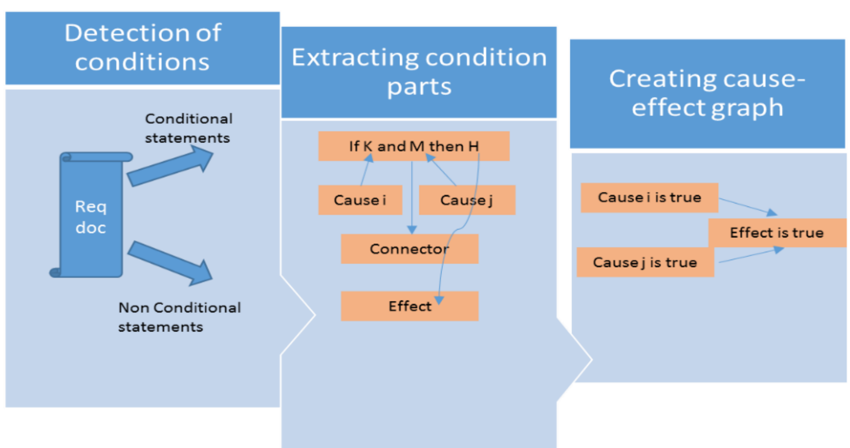

Fig. 3. Proposed system steps

3.1. Detection of Conditionals

The approach is implemented through a tool called CiRA (Conditionals in Requirements Artifacts) which automates the extraction of conditional logic from requirements and generates corresponding test cases. The process begins with the classification of natural language requirements. The system determines whether the requirement contains causal (conditional) statements or not. Requirements containing conditional logic (classified as "causal") are further processed, while non-causal requirements are ignored for test case generation. CiRA utilizes NLP techniques to scan the requirements documents and detect conditional sentences. This involves:

- **Tokenization**: Breaking down text into individual tokens (words, punctuation).
- **Part-of-Speech Tagging**: Identifying the grammatical roles of tokens (nouns, verbs, conjunctions).

- **Syntactic Parsing**: Analyzing the grammatical structure of sentences to identify clauses and phrases.
- **Pattern Recognition**: Recognizing common patterns associated with conditional statements (e.g., "If [condition], then [action]").

3.2. Extraction of Conditionals

For causal requirements, the system identifies the "causes" (conditions) and "effects" (outcomes) within the conditional statements. The causes or conditions are represented as top layer labels (e.g., "A is valid" and "B is false"), and the effects or outcomes are represented as bottom layer labels (e.g., "C is true"). Irrelevant information that does not contribute to the conditional logic is excluded. Once conditional sentences are detected, CiRA performs a detailed analysis to extract:

- **Antecedents (Causes)**: The "if" part of the conditional, representing the condition that triggers a particular outcome.
- **Consequents (Effects)**: The "then" part of the conditional, representing the expected system behavior or outcome when the condition is met.
- **Variables and Conditions**: Identifying the specific variables involved and the conditions applied to them within the antecedent and consequent.
- **Logical Relationships**: Understanding how multiple conditions are combined (e.g., conjunctions like "and," "or") to form complex conditions.

3.3. Creation of Cause-Effect-Graph

The process involves creating a Cause-Effect Graph (CEG) by mapping the identified causes and effects onto a visual representation. This graph visually illustrates the logical relationships between different elements of the requirements. The graph structure consists of nodes representing the causes (e.g., A and B) and edges linking the causes to the effects (e.g., c1, c2), leading to the resultant condition (e.g., C is true). This graphical representation aids in the systematic generation of test cases by clearly organizing the logical relationships.
The extracted antecedents and consequents are then mapped to a Cause-Effect Graph (CEG):

- **CEG Construction**: Nodes in the graph represent causes and effects, while edges denote logical relationships (e.g., AND, OR).
- **Logical Interpretation**: Conditionals can be interpreted as implications ($e1 \Rightarrow e2$) or equivalences ($e1 \Leftrightarrow e2$). CiRA supports both interpretations by allowing users to choose how conditionals should be treated, which affects the generation of positive and negative test cases.
- **Basic Path Sensitization Technique (BPST)**: CiRA applies BPST to traverse the CEG and identify the critical paths that cover all logical scenarios implied by the requirements. This technique ensures maximum coverage with a minimal number of test cases.

3.4. Generation of Acceptance Tests

The final step involves generating a set of acceptance test cases derived from the Cause-Effect Graph (CEG). Each test case is designed to represent a unique scenario based on the conditions and outcomes mapped in the CEG. The structure of each test case is organized into a table, where each row details the specific conditions (e.g., A, B) and the corresponding expected outcome (e.g., C). The approach ensures that both positive and negative test cases are generated, providing comprehensive coverage of all potential scenarios.

Results

To validate CiRA's effectiveness, we used a case study involving three industry partners (Allianz, Ericsson, and Kostal), applying CiRA to real-world requirements documents and comparing the generated test cases with those created manually. Example of the dataset (CiRA) is given in fig. 4 while a sample of the generated cause-graph from the dataset (CiRA) is given in fig. 5. An example of the list of test cases covering a specific requirement derived from the cause-effect graph is given in fig. 6.

This attribute is mandatory if the Material Safety Data Sheet attribute is "Y".
It is thus not represented in the model and the schemas.
It is used in conjunction with size group to completely define the size dimension.
This will allow for the size dimensions to be specified in using different size systems.
Business Rules: Either sizeDimension or descriptiveSizeDimension must be present, but not both.
The Retailers require this if the Item is marked as a Consumer Unit.
Only values from the enumerated list can be chosen from the UN/CEFACT.
Definition: A governing body that creates and maintain standards related to organic products./Brules: only registered values may be used.
If the Bar Code Type List class (telling which barcodes are on the package) exists, then there is a barcode on the package.
Business Rules: If the physical dimensions of the product change as a result of the promotion, then a new GTIN must be allocated.
If only a token is added to the product, then no need to change GTIN.
The code list required to identify the packaging material of the trade item. …………….

T1	VARIABLE 0 14	This attribute
T2	CONDITION 15 27	is mandatory
T3	KEYWORD 28 30	if
T4	VARIABLE 31 71	the Material Safety Data Sheet attribute
T5	CONDITION 72 78	is" Y"
T6	EFFECT_1 0 27	This attribute is mandatory
T7	EFFECT_1 31 78	the Material Safety Data Sheet attribute is" Y"
T8	NOT_RELEVANT 80 90	It is thus
T9	VARIABLE 137 139	It
T10	CONDITION 140 178	is used in conjunction with size group
T11	VARIABLE 200 218	the size dimension
T12	CONDITION 179 199	to completely define
T13	CAUSE_1 137 178	It is used in conjunction with size group
T14	EFFECT_1 179 218	to completely define the size dimension
T15	VARIABLE 220 224	This
T16	KEYWORD 225 239	will allow for
T17	VARIABLE 240 259	the size dimensions
T18	CONDITION 260 275	to be specified
T19	KEYWORD 276 278	in
T20	CONDITION 279 284	using
T21	VARIABLE 285 307	different size systems
T22	EFFECT_1 240 275	the size dimensions to be specified
T23	CAUSE_1 220 224	This
T24	CAUSE_2 279 307	using different size systems

Fig. 4. Sample of the requirements in the dataset (CiRA)

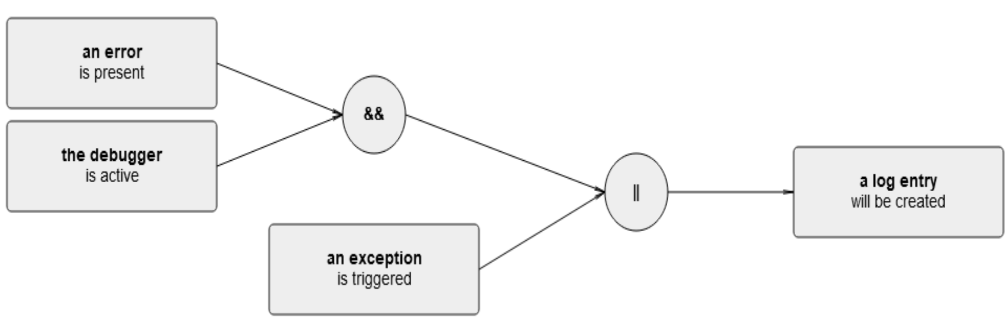

Fig. 5. Sample of the generated cause-graph from the dataset (CiRA)

id	an exception	an error	the debugger	a log entry
1	is triggered	not is present	is active	will be created
2	is triggered	is present	not is active	will be created
3	not is triggered	is present	is active	will be created
4	not is triggered	not is present	is active	not will be created
5	not is triggered	is present	not is active	not will be created

Fig. 6. Sample of test cases generated from the cause graph of a specific requirements in the dataset (CiRA)

Model Results:

CiRA successfully generated 71.8% of the manually created test cases across all datasets. It also identified 136 additional test cases that were missed in the manual process. The tool demonstrated the ability to handle complex conditionals and automatically generate a significant portion of the necessary test cases. Figure 7 illustrates the results of a case study comparing manually created test cases with those automatically generated by CiRA across three different companies: Allianz, Ericsson, and Kostal.

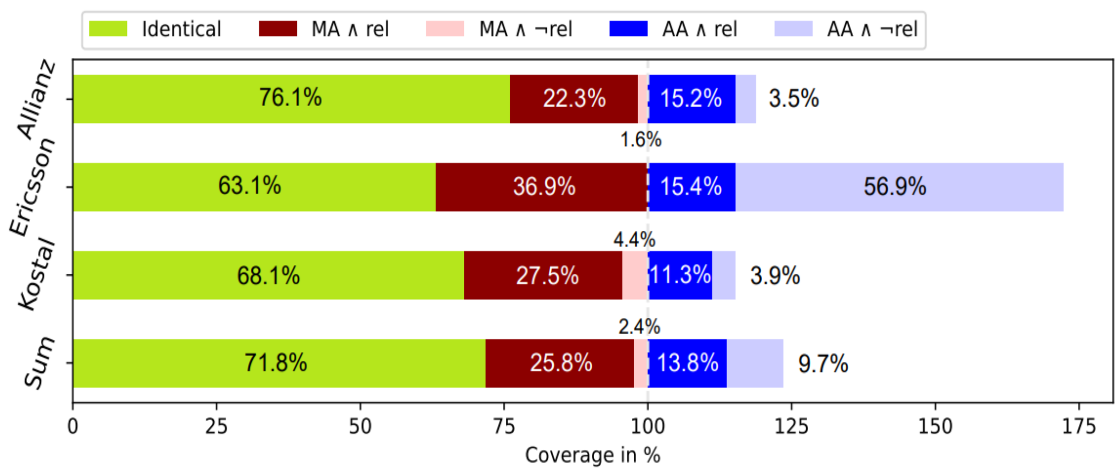

Fig. 7 . Proposed system results of a case study comparing manually created test cases with those automatically generated by CiRA across three different companies: Allianz, Ericsson, and Kostal.

Identical (Green): This represents the percentage of test cases that were identical between the manual and automatic methods. For example, at Allianz, 76.1% of the test cases created by CiRA were identical to the manually created ones.
MA ∧ rel (Dark Red): This indicates the percentage of manually created test cases that were necessary (relevant) but were not generated by CiRA. For Allianz, 22.3% of the manually created test cases were relevant and not automatically generated by CiRA.
MA ∧ ¬rel (Light Red): This shows the percentage of manually created test cases that were not relevant, meaning they were unnecessary and not included in CiRA's output. These cases were minimal, for instance, 1.6% at Allianz.
AA ∧ rel (Dark Blue): This represents the percentage of test cases that CiRA generated automatically, which were relevant but were missed in the manual process. Allianz had 15.2% of such cases.
AA ∧ ¬rel (Light Blue): This shows the percentage of test cases generated by CiRA that were not relevant, meaning they were unnecessary. Ericsson had a notably high percentage in this category (56.9%).

This figure highlights the strengths and weaknesses of CiRA in comparison to manual test case creation across different scenarios. It demonstrates that while CiRA automates a significant portion of the test case creation process, some manual oversight is still necessary to ensure all relevant cases are covered, and irrelevant cases are minimized.

Discussion

This analysis examines the results, emphasizing how the proposed approach enhances current practices for generating acceptance tests. By applying Natural Language Processing (NLP) techniques, the method significantly reduces manual effort and improves accuracy in generating test cases from textual requirements. The systematic extraction and utilization of conditional statements allow for more efficient test generation and comprehensive coverage of potential scenarios.

Nonetheless, the approach's effectiveness is constrained by several factors, particularly its dependence on the quality and precision of input requirements. Ambiguities or inconsistencies within natural language descriptions can diminish the accuracy of the generated test cases. Moreover, further testing on a broader range of datasets is essential to confirm the method's scalability and robustness across various domains and contexts.

Conclusion

The paper concludes that NLP can be effectively used to automate the creation of acceptance tests from software requirements. This approach offers significant advantages in terms of efficiency, accuracy, and coverage compared to manual test generation methods. The research also identifies potential areas for future work, including the refinement of NLP techniques for more complex requirements and the integration of this approach into broader software development practices. Additionally, The paper emphasizes the potential of CiRA as a tool to augment manual test case creation processes. While it does not fully replace the need for human involvement, particularly in understanding complex requirements, it significantly reduces the effort and increases the accuracy of test case creation. The authors suggest further research to improve CiRA's handling of complex conditionals and its integration into broader testing frameworks.

Acknowledgments

We are grateful to our colleagues at Allianz, Ericsson, and Kostal for their support and collaboration in this study. Special thanks are extended to the development teams involved for providing valuable insights and feedback that contributed to the refinement of the CiRA tool.

Disclosure

The authors report no conflicts of interest in this work. There are no financial interests or personal relationships that could have appeared to influence the research presented in this paper

References

1. Alagarsamy S, Tantithamthavorn C, Aleti A. A3Test: Assertion-Augmented Automated Test Case Generation. Inf. Softw. Technol. 2024; 176: Article ID pending.
2. CiRA: Causality in Requirements Artifacts. Blekinge Institute of Technology (BTH). Available from: CiRA: Causality in Requirements Artifacts (bth.se).
3. Frattini J. GitHub - JulianFrattini/cira: Python package for functions around the causality in requirements artifacts (CiRA) initiative. Available from: https://github.com/JulianFrattini/cira.
4. Frattini J. Zenodo: NLP-Based Requirements Formalization for Automatic Test Case Generation. 2021.
5. Frattini J, Goedicke M, Linsbauer L. Automatic creation of acceptance tests by extracting conditionals from requirements: NLP approach and case study. J Syst Softw. 2022.
6. Garousi V, Bauer S, Felderer M. NLP-assisted software testing: A systematic mapping of the literature. Inf. Softw. Technol. 2020; DOI: 10.1007/s00766-007-0054-0. Source: DBLP.
7. Giganto R. Generating Class Models through Controlled Requirements. New Zealand Computer Science Research Conference (NZCSRSC). Christchurch, New Zealand, 2008.
8. Kirinuki H, Tanno H. ChatGPT and Human Synergy in Black-Box Testing: A Comparative Analysis. arXiv 2024; 2401.13924.
9. Konrad S, Cheng BHC. Automated Analysis of Natural Language Properties for UML Models. 2010.

10. Lu G, Huang P, He L, Cu C, Li X. A New Semantic Similarity Measuring Method Based on Web Search Engines. WSEAS Trans. Comput. 2010; 9(1): Article ID pending.

11. Meziane F, Athanasakis N, Ananiadou S. Generating Natural Language Specifications from UML Class Diagrams. Requirements Eng. J. Springer-Verlag, London. 2013; 13(1): 1-18.

12. More P, Phalnikar R. Generating UML Diagrams from Natural Language Specifications. Int. J. Appl. Inf. Syst. Found. Comput. Sci. 2012; 1(8): Article ID pending.

13. Siricharoen WV. Ontologies and Object Models on Object-Oriented Software Engineering. IAENG Int. J. Comput. Sci. 2007; 33: Article ID pending.

14. Tichy WF, Landhabuer M, Korner SJ. Universal Programmability-How AI Can Help. Proc. 2nd Int. Conf. NFS Sponsored Workshop Realizing AI Synergies in Software Eng. 2013.