The Journal of Systems and Software 211 (2024) 112005

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Test case information extraction from requirements specifications using NLP-based unified boilerplate approach

Jin Wei Lim, Thiam Kian Chiew *, Moon Ting Su, Simying Ong, Hema Subramaniam, Mumtaz Begum Mustafa, Yin Kia Chiam

Department of Software Engineering, Faculty of Computer Science and Information, Technology, Universiti Malaya 50603, Kuala Lumpur, Malaysia

ARTICLE INFO

Keywords:
Natural language processing
Test case generation
Automation
Software requirements
Software Testing
Test Case

ABSTRACT

Automated testing which extracts essential information from software requirements written in natural language offers a cost-effective and efficient solution to error-free software that meets stakeholders' requirements in the software industry. However, natural language can cause ambiguity in requirements and increase the challenges of automated testing such as test case generation. Negative requirements also cause inconsistency and are often neglected. This research aims to extract test case information (actors, conditions, steps, system response) from positive and negative requirements written in natural language (i.e. English) using natural language processing (NLP). We present a unified boilerplate that combines Rupp's and EARS boilerplates, and serves as the grammar guideline for requirements analysis. Extracted information is populated in a test case template, becoming the building blocks for automated test case generation. An experiment was conducted with three public requirements specifications from PURE datasets to investigate the correctness of information extracted using this proposed approach. The results presented correctness of 50 % (Mdot), 61.7 % (Pointis) and 10 % (Npac) on information extracted. The lower correctness on negative over positive requirements was observed. The correctness by specific categories is also analysed, revealing insights into actors, steps, conditions, and system response extracted from positive and negative requirements.

1. Introduction

Testing, in the context of a software product called software testing, is the process of determining whether the system is producing accurate results for various inputs and meets all the requirements. It is a crucial task in guaranteeing software quality and aids in identifying potential application bugs (Salam et al., 2022). IEEE Std 829 specifies all the stages of software testing and documentation at each stage. According to IEEE Std 829, a test case is a set of inputs, conditions, and expected results for systems under test (IEEE Standard for Software and System Test Documentation, 2008). Testing is an expensive task which takes 40 %–60 % of the time, cost, and effort (Mustafa et al., 2021); and hence, automated testing has been proposed. Researchers such as Wang et al. (2020) and Aoyama et al. (2021) proposed strategies that require minimal human interventions to generate test cases from requirements written in Natural Language (NL). However, requirements specified in NL contain ambiguities, inconsistencies, and incompleteness that influence the quality of software testing. To avoid ambiguity, requirements are often defined in a formal or semi-formal way (e.g., UML, Finite State Machine). Nevertheless, this formal model approach is not widely adopted in test automation due to high cost and complexity to develop and maintain the formal models (Mai et al., 2018). Additionally, Carvalho et al. (2014) also suggested that it is challenging to use formal models to communicate with non-technical staff who have limited knowledge of formal specifications. The training required to implement a formal approach is also time-consuming and expensive. Thus, in the software development industry, NL is still widely used and applied to various problem domains.

The use of Natural Language Processing (NLP) techniques to discover and extract information from textual descriptions written in NL is rather prevalent in the requirements engineering research. Tiwari et al. (2019) proposed a systematic transformation approach using NLP to automatically extract use case elements from textual problem specifications through an analysis process with NLP techniques. Aoyama et al. (2020) also proposed an NLP approach that analyses and transforms requirements written in NL into semi-structured form before converting

* Corresponding author.
E-mail address: tkchiew@um.edu.my (T.K. Chiew).

https://doi.org/10.1016/j.jss.2024.112005
Received 4 September 2023; Received in revised form 29 January 2024; Accepted 12 February 2024
Available online 17 February 2024

the information into a decision table. This approach by Aoyama et al. (2020) uses a semi-formalizer and multiples grammatical rules to extract the logical relation in requirements into semi formal descriptions.

Despite the attempts to use NLP in requirements research, the inherent ambiguity and inconsistency in requirement statements caused by NL are widely acknowledged in the works of Fischbach et al. (2019), Tiwari et al. (2019) and Mustafa et al. (2021). This substantially increases the challenges of extracting test case information. Detecting ambiguities in NL is relatively hard, as a statement written in NL may permit multiple interpretations. One of the main causes of software project failures is inefficiency in sorted out requirements written in NL, such as the meaning of phrases in the specifications (Mustafa et al., 2021). Additionally, negative requirements, which are described as what the system should not perform in a work done (Mai et al., 2018), also tend to introduce ambiguities and are frequently neglected. Boilerplate, or more commonly known as templates, enables the structuring of a sentence's syntactic structure into predefined segments (Arora et al., 2015). It allows an efficient way to reduce ambiguity and avoid inconsistency in requirements in natural language and has garnered engineers' attention as they can minimise the vagueness when writing requirements in natural language. In this study, we utilised a boilerplate (Section 4.1.1) as a template and guideline to extract useful information (i.e. actors, conditions, steps, system response) from requirements that can be used to form test cases for the relevant requirements.

To maximise the benefits of using boilerplates for software requirements analysis, it's essential to employ well-designed templates that cover a variety of natural language patterns. Comprehensive boilerplates ensure accurate information extraction from diverse requirement statements. Hence, the way requirements are written affects automation of test case information extraction. Software requirements can be written in positive or negative manners to ensure the overall coverage of the system's functionalities. When written in a positive manner, positive requirements describe what users are expecting and the desired behaviour of the software system. When written in a negative manner, negative requirements describe the undesirable behaviour of the system (Mai et al., 2018). However, negative sentences often lead to ambiguity and are usually paraphrased manually or overlooked in existing research works on automatic extraction of test information from functional requirements (Shweta and Sanyal, 2020).

The contribution of this study lies in addressing the challenges associated with ambiguities and inconsistencies in software requirements for test case information extraction. Concerning ambiguities and inconsistencies in requirement statements, this study extracts useful test case information from software requirements written in NL (i.e. English), minimising human interventions to reduce the time and effort compared to the conventional methods. This research uses a unified boilerplate as a guideline on the structure of the requirements to analyse the requirements instead of limiting the ways of writing the requirements. The unified boilerplate covers both positive and negative requirements. Correlation between the utilisation of the unified boilerplate approach and the requirements was studied. Additionally, the approach was applied on three sets of open source requirements. The correctness of the extracted test case information was examined, too. The incorporation of cross-validation in the evaluation not only ensures inter-rater reliability but also provides valuable insights on our approach.

The rest of the paper is organised as follows: Section 2 explains the background of this study; Section 3 mentions the related works. Section 4 explains the proposed approach and Section 5 describes the evaluation result. A discussion of evaluation results is presented in Section 6. Finally, Section 7 concludes this study and discusses future work.

2. Background

This section describes the background of this study which includes test case elements, the NLP tool and requirements boilerplates. Besides, the section also discusses existing work related to this study.

2.1. Test case elements

A test case is defined as "a specification of inputs, execution conditions, testing procedure, and expected results" (Hue et al., 2019). It is a set of commands or instructions that a tester needs to execute to verify a specific feature of a product or application. A test case document usually includes actors, test steps, preconditions and postconditions that verify requirements. The actor is typically located in the noun segment of the sentence, while the main action verb can be identified in the action segment (Gröpler et al., 2021). This information can be extracted from requirement specifications but the completeness of a test case relies heavily on the information present in each requirement statement. As for test input data, the information in requirement specification alone might not be sufficient to generate comprehensive input data. So, a tool is needed to understand NL and extract test case information from requirements written in NL.

2.2. Natural language processing tool

NLP tools are commonly used to perform classification, tokenization, stemming, tagging, parsing, and semantic reasoning on NL. A NLP pipeline that executes multiple analyses (e.g. tokenization, POS tagging, chunking and semantic role labelling) has been introduced by Tiwari et al. (2019) to automate textual requirements analysis and populate use case scenarios for test case generation. This NLP approach is well known and has also been adopted by several researchers including Aoyama et al. (2020), whose approach transforms requirements into semi-structured form with a semi-formalizer to generate test cases. While their algorithm incorporates NLP techniques and decision tables, it is based on predefined keywords and thus lack of keywords has caused errors when handling complex logical relations defined using terms that are not in the dictionary. Natural Language Toolkit, commonly known as NLTK, is a suite of text processing tools and libraries with various NLP techniques (Loper and Bird, 2002). NLTK offers user-friendly interfaces to more than 50 corpora and lexical resources, including the extensive WordNet. This toolkit includes a comprehensive suite of text processing libraries that facilitate tasks such as classification, tokenization, stemming, tagging, parsing, and semantic reasoning. In addition, NLTK provides wrappers for robust industrial-strength NLP libraries. It serves as a go-to resource for researchers, educators, and developers working on computational linguistics. With the NLP techniques and ready-to-use libraries, our approach can utilise these NLP tools to analyse the requirements written in NL.

2.3. Requirements boilerplates

Requirements boilerplates, which is a NL pattern that limits the syntax of sentences to a predefined linguistic structure, have long been considered to be effective strategies for writing requirements and reducing ambiguity in NL requirements (Arora et al., 2014). According to Tiwari et al. (2022), boilerplates or templates are widely adopted as they have correlated and straightforward syntax which facilitates the translation process using NLP techniques. They propose Rupp's and EARS boilerplates for translating requirements into templates. Fig. 1(A) and 1(B) illustrates the boilerplates : (A) Rupp's boilerplate, (B) EARS boilerplate. Research from Arora et al. (2015) which involves verifying adherence to requirements boilerplates, also uses Rupp's and EARS boilerplates. It has motivated the combination of both boilerplates in our research, which will be explained further in Section 4.1.1.

3. Related work

There are other works that extract test cases information automatically from requirements written in NL. Researchers have been discussing

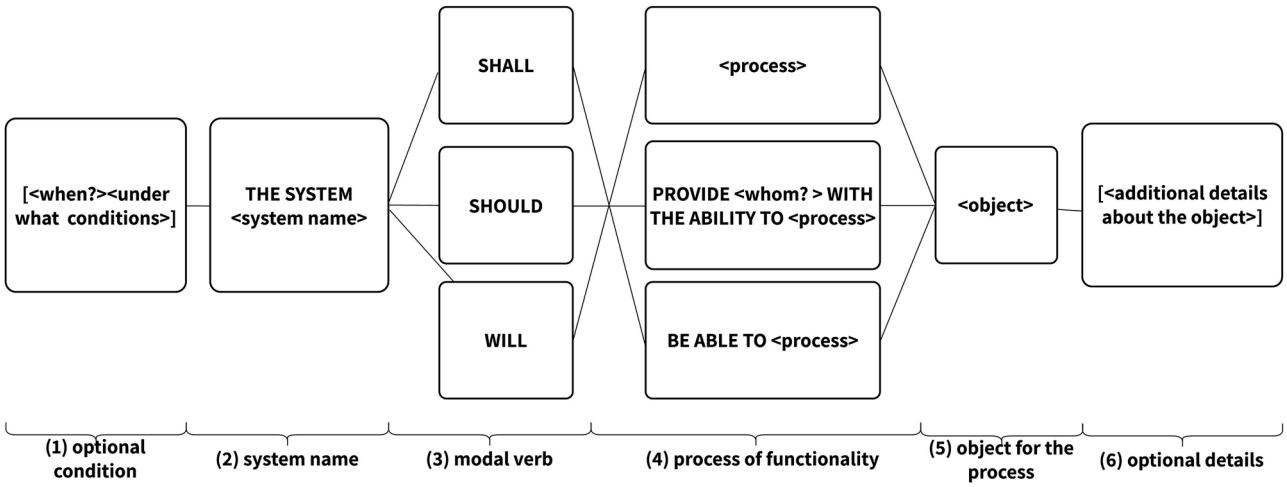

Fig. 1A. Rupp's boilerplate.

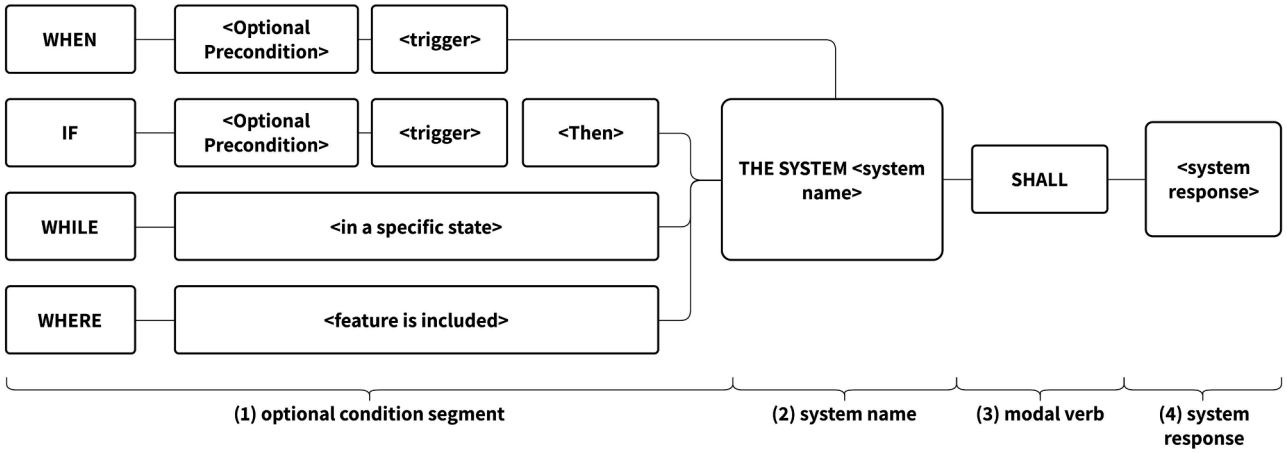

Fig. 1B. EARS boilerplate.

the issues with requirements written in natural language such as requirements specifications, use case specifications and semi-structured specifications. One proposed approach is model-based testing (MBT) that uses a behavioural tree requirements model, which is a graphical notation for requirements capture to generate test cases (Lindsay et al., 2015). Their approach outperforms other existing approaches in generating test cases by finding representatives of all possible execution paths in a behaviour tree model.

Aoyama et al. (2021) also introduce a test design process that uses semi-structure models to automate the test case generation in the domain of Software Product Line Engineering (SPLE). The studies confirmed that methods that use models can reduce the time for design and execution compared to the conventional manual routine. Model-based testing strives to make testing more agile, less costly, and of greater quality. However, it may not be feasible when it comes to complex and real situations for a complete automated process (Carvalho et al., 2014).

Some researchers also proposed approaches that increase the accuracy through human intervention during the test case generation process. Wang et al. (2020) proposed a method which requires human inspections of the generated Object Constraint Language (OCL) and making corrections when needed, before generating test cases. With this approach by Wang et al. (2020), manual human effort has been reduced to minimal as the algorithm can achieve high correctness of 96 % in generating OCL constraints for test case generation. A research by Tiwari et al. (2019) also uses questionnaire-based approach and NLP to derive use case scenarios from requirements. Their approach involved human validation to refine and improve the output quality with a series of questionnaires to be answered by humans. Aoyama et al. (2020) also uses repetitive reviews by engineers manually to eliminate errors in test cases. While their proposed method used semi-formal descriptions to depict flaws and promote error detection, it still needs human intervention in the test case generation process . When the test case generation process relies on human intervention, scalability becomes an aspect that needs to be investigated as it could become a limitation. The effort and time for human intervention could increase drastically as the number of specifications increases, potentially affecting the correctness of extracted information.

There is also research that proposed a restricted requirements template with limited keywords and specific rules to reduce ambiguity in requirements and increase the precision of information extraction (Wang et al., 2020). A research conducted by Mai et al. (2018) proposed an approach to generate test cases by having a set of misuse case specifications which are elicited according to a template that includes certain keywords to support the information extraction process. They have proved the feasibility of their approach by evaluating the effectiveness and effort needed in discovering vulnerabilities of a system from both positive and negative requirements. However, their approach has limited scope of applications as it only concentrates on addressing security requirements. Carvalho et al. (2014) also proposed NAT2TEST that uses controlled natural language (SysReq-CNL) in writing requirements to avoid misinterpretation and ambiguity. Restricting the

ways of writing requirements might increase the correctness of the test case generation process, but it poses challenges for analysing existing requirements that are not written according to the predefined rules. It also requires additional knowledge acquisition by the engineers to write requirements in a more restricted language with restricted expressiveness.

Another proposed approach is using a semi-formal structure in the process of test case generation from natural language requirements to reduce ambiguity and clarify the logical relations such as and, or, not and imply, among the atomic propositions in the requirements. Aoyama et al. (2020) found that the logical relations can be depicted explicitly using a semi-formalizer in the algorithm, but the complex relations still pose difficulties to the algorithm. Fischbach et al. (2019) also proposed a method that identifies and interprets the semi-structured requirements, which is a "pseudo-code" description in the requirements specifications to understand the logical constructs. Their approach uses requirements from the PURE dataset by Ferrari et al. (2017) and was evaluated based on time saved. It demonstrated a time savings of 86 %. The proposed semi-structured requirement, similar to restricted NL, uses a control structure to avoid ambiguity in NL. This method provides better flexibility in writing requirements as it does not have a rigid syntax and limited vocabulary compared to controlled natural language that restricts the writing of requirements with specific keywords or notations.

Concerning these existing approaches, there are some gaps to further improve the automated test case generation process. Some of the work mentioned need human intervention in the process and other requires a strictly controlled language for writing requirements. Table 1 summarises various methods used in other work and the tools used. Comparing the other work, our research uses boilerplates as a guideline on the structure of the requirements to analyse the requirements instead of limiting the ways of writing the requirements. The boilerplate derived and used in this approach also accepts negative requirements, which most of the works mentioned only consider positive requirements or domain-specific requirements.

4. Unified boilerplate approach

This section describes an overview of the proposed approach. The methodology used for sentence structure analysis, the analysis process to extract information from the pre-processed requirements, and the generation of test case information are presented in Sections 4.2, 4.3 and 4.4, respectively.

4.1. Approach description

An overview of the approach is shown in Fig. 2, which involves three phases of automated process and one manual evaluation process. In this proposed approach, software requirements are analysed according to the unified boilerplate shown in Fig. 3, which is a combination of Rupp's boilerplate and EARS boilerplate.

The first phase focuses on the pre-processing of NL requirements using NLTK, a NLP toolkit by Bird et al. (2009), as the tool for NLP processes. The requirements undergo a pipeline of NLP tasks to enhance their quality, remove ambiguity such as syntactic and negation ambiguities, and interpret the meaning of each word in the requirement. This pre-processing phase ensures that only relevant information is retained, while unrelated data (such as explanations or examples) is removed to improve the accuracy of test case information extraction in the subsequent phases.

The second phase involves extraction of useful test case information from the pre-processed requirements based on the unified boilerplate. Firstly, by using the unified boilerplate approach, actors are identified in each sentence of the requirements by recognizing human nouns based on a dictionary of English words. Synset interface in NLTK is used to look for related words in WordNet, a lexical database for English. WordNet organises nouns, verbs, adjectives, and adverbs into sets of cognitive synonyms, known as synsets, with each synset representing a unique conceptual idea (Miller, 1994). For cases where no human noun exists in the requirements, the algorithm by default will use the "default user" as the actor. The algorithm also identifies the conditions by matching predefined strings (i.e., when, where, while, if) to the requirements. In addition, the algorithm uses string search techniques to determine if a requirement is a positive or negative requirement based on some predefined negative keywords (e.g. not, prevent).

In the third phase, detailed descriptions of the sequential actions, described as test steps, are generated by recognising the verbs and objects in the requirements. The test case information extracted is automatically populated in a template to produce a document that consists of a collection of test case information.

An experiment was conducted with three sets of requirements specifications from the PURE dataset, a dataset of 79 publicly available natural language requirements documents collected from the Web (Ferrari et al., 2017). These requirements were downloaded from online sources in XML file format. Mdot, Pointis and Npac requirements were used to investigate the correctness of the information extracted using the proposed approach. In the evaluation phase, the obtained results were evaluated by multiple evaluators to determine the correctness of the information extracted. The generated test case information from every

Table 1
Summary of methods and tools used in related work.

Method	Research	Tools	Purpose
MBT	Lindsay et al. (2015)	Behavioural tree model	Generate functional test cases from requirements written in NL using model notations and confirmed by model checking.
	Aoyama et al. (2021)	Semi-structured model	Convert specification written in NL to semi-formal description for generating test cases.
Human intervention	Tiwari et al. (2019)	Questionnaire	Extract use case elements from the textual requirements specification which can be used in test case generation.
	Wang et al. (2020)	Inspection	Capture information from requirements specifications in NL automatically for manual inspection to generate acceptance test cases.
	Aoyama et al. (2020)	Reviews	Convert specifications written in NL to test cases by visualising the logical relations for engineers to review.
Semi-formal structure		Logical relations	Convert specifications written in NL into semi formal descriptions with explicit logical relations to generate test cases.
	Fischbach et al. (2019)	Pseudo-code	Automatically identifies semi-structured requirements descriptions written in NL and translates them into a test model.
Restricted requirements	Carvalho et al. (2014)	Controlled natural language	Generate test cases from requirements written in controlled NL that are standardised.
	Mai et al. (2018)	Misuse case specification	Generates security test cases automatically from misuse case specifications that are written according to a template.
	Wang et al. (2020)	Controlled natural language	Extract behavioural information from requirements written in NL according to a template with keywords and restriction rules to generate test cases.

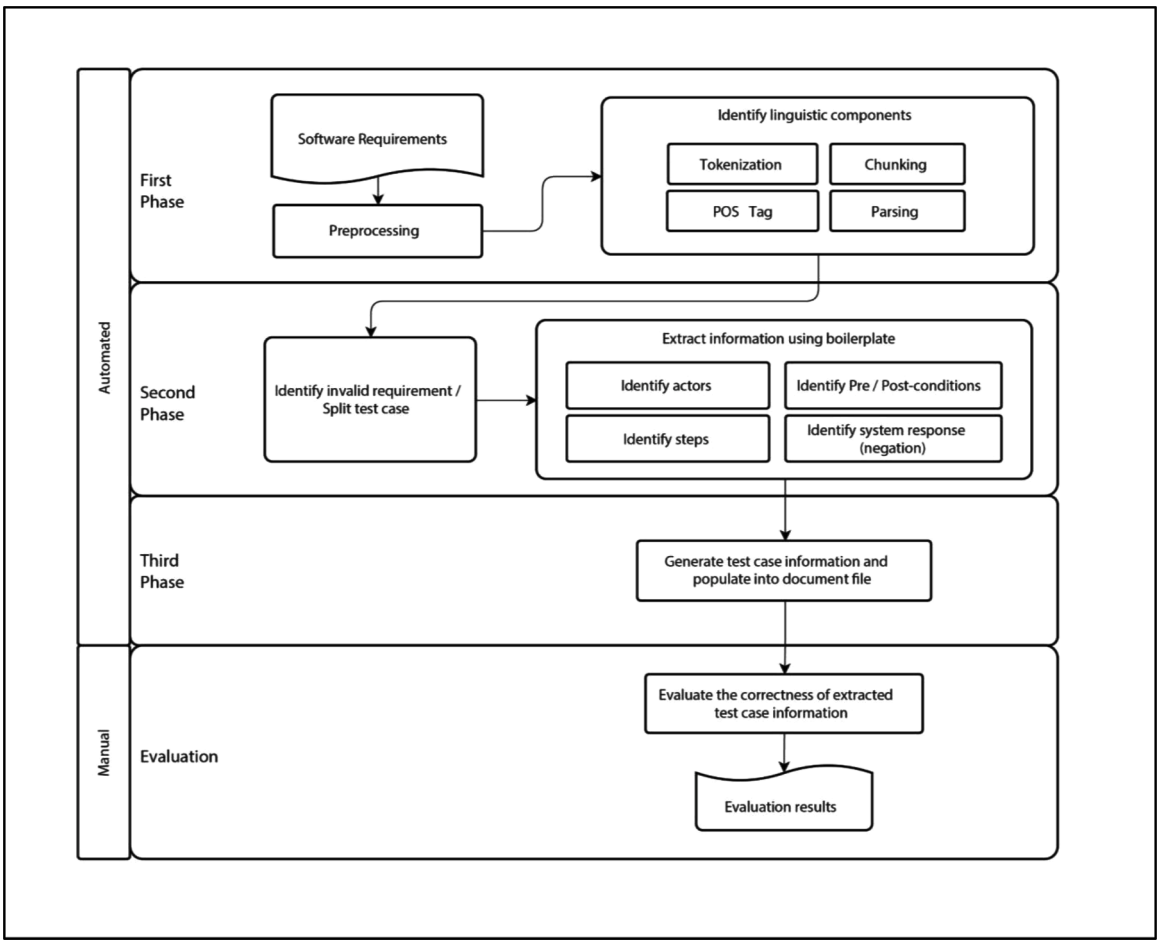

Fig. 2. An overview of the proposed approach.

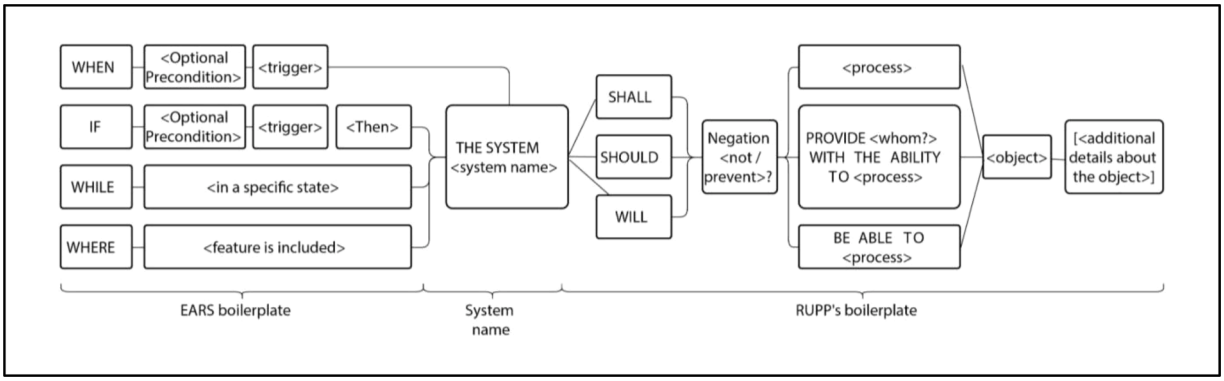

Fig. 3. Requirements boilerplate that combined Rupp's Boilerplate and EARS Boilerplate.

requirement dataset was cross-validated by two evaluators independently according to different categories (e.g. actor, conditions, steps, system response). Disagreements among evaluators were resolved through discussions and mutual agreements to ensure inter-rater reliability. Detailed explanations of each phase are provided in the subsequent subsections.

4.1.1. *Unified boilerplate*

The implementation of EARS and Rupp's boilerplates in the previous research by Arora et al. (2015) and Tiwari et al. (2022) has inspired our research to combine and derive boilerplates. There are six segments in Rupp's boilerplate: (1) optional condition; (2) system name; (3) modal verb (shall/should/will); (4) process of functionality; (5) object for the process and (6) optional details about the object. In this boilerplate, each segment of a sentence is clear and can be distinguished. Another well-known boilerplate is EARS boilerplate (Mavin et al., 2009). This boilerplate is made up of four segments : (1) optional condition; (2) system name; (3) modal verb and (4) system response. EARS boilerplate emphasises more on specifying conditions with more advanced condition segments in the boilerplate, while Rupp's boilerplate has a more elaborated structure on other segments. By combining both boilerplates to a unified boilerplate, a more inclusive boilerplate that represents each segment in the requirement can be used as a guideline for analysing requirements written in NL. Fig. 3 shows the unified boilerplate,

combining Rupp's boilerplate and EARS boilerplate. In this unified boilerplate, EARS boilerplate makes up the front part of the combination, which caters for the conditions and followed by Rupp's boilerplate for the actor and action segments for test steps.

4.2. First phase - Pre-processing

In this phase, the requirements are first pre-processed automatically, to remove ambiguity and to increase the quality of the result. Repetitive and insignificant words, known as stop-words (e.g. a, an, the), are eliminated. Requirements with multiple sentences are separated, and each sentence is individually processed through a pipeline of NLP processes. This pipeline includes tokenization, Part-of-Speech (POS) tagging, chunking, and parsing, which collectively transform each requirement statement into useful information. POS tagging is used to assign and label each word element in the requirements with their part-of-speech tags (e.g. VB for verb, NN for singular noun). Parts of speech include nouns, verbs, adverbs, adjectives, pronouns, conjunction, and their subcategories. It then uses POS tags as input for syntactic analysis, also known as the chunking process. This research emphasises regular expression-based chunking. The chunker determines the actor, modal and action in the requirement based on a predefined regex pattern. The chunk pattern is shown in Table 2. This approach can provide an abstraction for each segment such as the actor segment and action segment in a requirement statement, also known as chunks. 'CLAUSE' chunk is used to mark the verbs when there are two actions in a requirement statement. It will be used for further processing such as splitting in the second phase, as shown in the algorithm in Algorithm 1. The segment identified between actor and action is labelled as 'SEP' chunk to allow the information at different positions of the requirement statement to be analysed accurately. The patterns of POS tags combination is based on the derived boilerplate to cater for different patterns of sentence structure.

Basically, chunking is the main technique used in this approach to identify different segments of the requirement statements. Fig. 4 and 5 display examples of different segments in a requirement. In Fig. 4, the requirement starts with a condition ("If security level is 0"), then an actor ("user"), followed by the modal verb segment ("shall be able to") and its action ("switch") that makes up the action segment. The last segment, which is the object segment, consists of the objects ("predefined structure list") the action is referring to. Fig. 5 shows another example with negation. The modal verb segment is amended to "shall not be able to". In this example, the "not" word signifies negation in this requirement and the system shall respond adversely by disallowing the specified action.

4.3. Second phase - Analysis Process

In this phase, the algorithm follows extraction rules to extract information from the pre-processed requirements. The unified boilerplate is used in this phase to efficiently identify the sentence segments in the requirement. It is used as a guideline on the sentence structure and the patterns of chunks and POS tags for each segment. Rupp's and EARS boilerplates are combined and integrated together in this approach. The boilerplate comprises distinct segments, including the actor segment, action segment, object segment and an optional condition and negation segment. Identification of actor and action segments relies on the POS tags and chunks identified during the pre-processing phase while conditions are based on the predefined key-phrases (i.e., when, where, while, if). System response relies on the identification of negation in the requirements.

In the derived boilerplates, an optional condition can appear before the actor. To correctly extract the condition segment, the algorithm will identify the positions of the actor, modal (e.g., shall) and action from the chunked requirement as the anchors. Punctuations are not considered as a start or end indicator of the condition in the derived boilerplate to reduce the dependency on the comma, which is not reliable as it could be different writing styles from different requirements engineers. The boilerplate also has an optional negation segment between the actor and action for negative requirements.

Table 2
Chunk pattern.

Chunk Name	POS Tags Combination	Description
CLAUSE	<VB|VBG><CC><VB|VBG>	Two actions exist in a requirement statement (verb AND/OR verb)
SEP	<MD> <MD><VB>?<JJ>?<IN> <MD><VB><JJ><TO> <MD><VB><NN>*<TO> <MD><JJ><TO> <MD><VB><JJ><TO>	Segment that appears between actor and action

MD: modal verb, NN : noun, singular, CC: conjunction, VB: verb, base form, VBG: verb, present participle, JJ: adjective, IN : preposition/subordinating conjunction, TO : infinite marker (to), ? : optional, * : one or more, | : or.

4.3.1. Identify invalid requirements for test cases

Each input statement must have an action and modal verb in it to be considered as a valid requirement. The algorithm will determine the presence of any action in the requirements by looking for verb labels in the POS tags (e.g., VB, VBG). Missing modal verbs (e.g., should) in the requirement are also indicators for invalid requirements. Modal verbs are identified by searching for separation tags (e.g., SEP) in the chunked requirements. Requirements with no modal verbs or actions will be deemed invalid and will be ignored from further processing. As an example, "*The system shall eliminate any and all dependencies on workstation (user) control/parameter files for program operation. For example, parameter information required by the NBI Translator (including information currently stored in ELEMENTS.PRN and ERRORS.LST) will be stored strictly within the Pontis database rather than on individual user workstations.*", after splitting this requirement into separate sentences, the second sentence which provides supplementary example information to the first sentence is invalid. Therefore, some input requirements may not have the corresponding output test case information.

4.3.2. Split test cases

A requirement with more than one action should be split into separate test cases. This allows each test case to focus on a dedicated test item. In the chunking process, the pattern of multiple actions in one requirement will be chunked and labelled as a custom clause to be recognised by the algorithm for splitting it, as shown in Algorithm 1. To avoid creating vague sentences during the splitting process, the algorithm will only split actions that are referring to the same object. Taking an example requirement from a Bridge Management System in PURE dataset (Ferrari et al., 2017) : "The user shall be able to edit and remove existing structures." will be split into two test cases with two actions which are edit and remove respectively.

4.3.3. Identify actor

Based on the derived boilerplate, the actor of the requirements may appear in several positions. The identification of an actor's position is crucial as it determines the extraction process of other information. The actor segment is usually positioned before the modal verb and in between the modal verb and action in some cases. For cases of ⟨ACTOR⟩ ⟨MODAL-VERB⟩ ⟨VERB⟩ ⟨OBJECT⟩, the first actor segment will be recognised. However, for cases of ⟨FIRST-ACTOR⟩ ⟨MODAL-VERB⟩ ⟨VERB⟩ ⟨SECOND-ACTOR⟩ ⟨TO⟩ ⟨VERB⟩ ⟨OBJECT⟩, such as "The DUAP System shall allow the system administrator to add, modify, and delete Output Services.", "system administrator" will be selected as the actor. The second actor segment will be prioritised if it exists according to the boilerplate.

Algorithm 1
Algorithm for preprocessing and identify actor.

```
while not end of the sentence do
    Tokenize each word
    Tag Part-Of-Speech to the word
  end while
  Chunk and Parse the tagged sentence
  if has pattern ⟨VERB⟩ AND/OR ⟨VERB⟩ then
    Split the actions into separate sentences
  end if
  for each sentence do
    Identify Part-Of-Speech with Noun
    for each noun do
      Search with Synset in WordNet for human noun
  end for
  end for
```

If security level is 0	The user	shall be able to	switch	between predefined structure lists.
Condition segment	Actor Segment	Modal Verb	Action Segment	Object Segment

Fig. 4. Segments in requirement without negation.

If security level is 0	The user	Shall not be able to	switch	between predefined structure lists.
Condition segment	Actor Segment	Modal Verb	Action Segment	Object Segment

Fig. 5. Segments in requirement with negation.

After identifying the actor segment, the algorithm will select the noun by recognising the POS tags (e.g., NN). Each noun in the segment will be processed to determine whether it is a "human noun", which is a noun that relates to a person or individual, by using synset in WordNet to look up for words in the 'person' category. By default, if there is no human noun identified from the requirements, "default user" will be used as the actor. Take as an example the requirement: "The user shall be able to switch between predefined structure lists.", "user" is the only noun in the actor segment. Another example requirement: "The DUAP System shall collect probe vehicle data.", "DUAP System" is identified as the actor segment and system is considered a noun. However, the system is not a human noun so "default user" will be used as the actor in our algorithm. Fig. 6 shows the different positions of actor segments in the requirement statements.

4.3.4. *Identify condition*

A precondition refers to a condition or predicate that must be true before a method can be successfully run. A postcondition is a condition or predicate that can be guaranteed after a method is finished. The conditions (preconditions or postconditions) are extracted by identifying certain keywords in the written requirements. A collection of keywords (i.e., when, where, while, if) is used by the algorithm to search

Requirement
The {user} shall be able to switch between predefined structure lists.
The DUAP System shall allow the {system administrator} to add, modify, and delete Output Services.
{} represents actor segment

Fig. 6. Example of actor segment.

for the presence of a condition. According to the derived boilerplate, the position of a condition can appear at the beginning of the requirements, which is before the modal verb (e.g., must). For example, "The user shall have the option to copy the notes from the previous inspection when creating a new inspection.", the presence of the conditional keyword "when" represents a condition in this requirement.

4.3.5. Identify actions for steps

The modal verb in the requirement influences the starting point (also known as *start_index*) where the algorithm starts identifying the steps. The chunking process will chunk the separation segment based on the presence of a modal verb. It uses chunk labels and POS tags to match the regular expression patterns. To correctly identify the steps in the requirement, the position of *start_index* will always be after the actor's segment as the actor might appear at several positions. Once the *start_index* is determined, the algorithm will look for verbs that represent the action by searching the POS tags (e.g., VB). The segment after the verb will be identified as the object referring to that action. By lemmatizing the words with NLP tools, the lemmatized verb can be generated to form a valid sentence that represents the steps from extracted verb and object. For example, in this requirement "The Disputes System must prevent users from accessing any dispute cases that do not belong to their cardholder base.", the word "accessing" is the action and will be lemmatized to its base form, "access", to generate grammatically accurate test steps. Fig. 7 shows some examples of start_index.

4.3.6. Determine positive or negative response

In this process, we defined "not" and "prevent" in the list that represents negation. For negative requirements, the steps will proceed with the undesirable actions and the system shall not allow the action. A requirement will need to be checked for the presence of any negation word. Based on the derived boilerplate, these negation words will exist in the action segment and be positioned before the verb. If the position of the negation word identified is after the verb, it will not be considered. The negation word will usually appear in between the actor and the action, and next to the modal verb (e.g., should not). Once the requirement has been identified as a negative requirement, a negative response (i.e., system not allow) will be assigned to this test case when populating the information, or else a positive response (i.e., the system allows) will be assigned to the positive requirements. As an example, "The product shall prevent the player from overlapping ships on their grid.", the word "prevent" that appeared in this requirement represents negation and negative response shall be assigned to this test case.

4.4. Third phase - Generation of test case information and population of the test case template

After the extraction process, a visual representation is generated to display all the information in a structured and understandable form. A test case information template (shown in Fig. 8) is used to present the information extracted from the requirements according to the items in the template. The items in the template are populated with the information identified from the analysis process in the second phase such as placing actors in the actor column, listing test steps in the steps column, and specifying conditions in the precondition column. Fig. 9 displays an example of information in the test case template populated from the requirement "The user shall be able to switch between predefined structure lists.".

4.5. Data sampling

Three sets of requirement specifications were obtained from PURE Datasets, a dataset of publicly available NL requirements documents collected from the Web (Ferrari et al., 2017). These datasets have also been used in other studies such as work done by Fischbach et al. (2019) to identify semi-structured requirements and interpret the logical constructs from the requirements specifications. The way of writing for the selected requirements closely, but not exactly, resembles the derived boilerplate, with no augmentation. All requirements analysed were functional requirements and non-functional requirements were excluded. The requirements specification details are shown in Table 3, PR denotes positive requirement and NR denotes negative requirement.

Dataset 1: Pointis, a functional requirement specifications for bridge management system prepared by Cambridge Systematics Inc.
Dataset 2: Mdot, a system requirement specifications for Michigan Department of Transport's Data Use Analysis and Processing (DUAP) System.
Dataset 3: Npac, a functional requirement specification for Number Portability Administration Center (NPAC) Service Management System (SMS).

5. Result of experiment

5.1. Design and evaluation of experiment

In this research, three requirement specifications, which is Pointis, Mdot and Npac from PURE Dataset by Ferrari et al. (2017), were used to study the correctness of the result generated by the proposed approach. For each dataset, the correctness of the generated test case information was independently validated by two evaluators, who possessed Ph.D. qualifications and lecturing in the field of software engineering for more than 10 years. This diverse and knowledgeable group of evaluators contributes to the credibility and reliability of the assessment. An evaluation template is used to categorise the different types of errors in extracted test case information. It is utilised by the evaluators to verify the generated results from higher level (e.g., actor and steps), to more specific breakdowns within the category (e.g., actor not identified or actor wrongly identified). The extracted information is considered correct when there are no errors in any of the evaluated categories. Table 4 displays the evaluation categories and their respective breakdowns. To establish inter-rater reliability, the evaluations were also cross-checked and those conflicting evaluations were resolved through discussions. Facilitated meetings were carried out between evaluators to achieve mutual agreement. The evaluation process was conducted in two rounds. During the first round, feedback and insights from both

Requirement
The user >> shall be able to switch between predefined structure lists.
The DUAP System shall allow the system administrator >> to add, modify, and delete Output Services.
>> represents start_index

Fig. 7. Example of start_index.

Test Case ID	
Actor	
Pre-condition	
Steps	
Expected Outcome	

Fig. 8. Example of test case information template.

Test Case ID	T-001
Actor	user
Pre-condition	-
Steps	1. user switch between predefined structure lists.
Expected Outcome	System allow

Fig. 9. Example of populated test case information.

Table 3
Requirements dataset information.

#Dataset	Requirement Specification	No. of PR	No. of NR
Dataset 1	Pointis	167	–
Dataset 2	Mdot	142	–
Dataset 3	Npac	–	29

PR : Positive requirement, NR : Negative requirement.

Table 4
Evaluation categories and their respective breakdowns.

No.	Evaluation category	Evaluation result
1.	ACTOR	Not Identified Wrongly identified Partially identified
2.	PRE/POST CONDITIONS	Not Identified Wrongly identified Identified but with ambiguous sentence
3.	STEPS	Not Identified Wrongly identified -incorrect steps Wrongly identified -missing steps Wrongly identified -additional steps Wrongly identified -incorrect sequence Identified but with ambiguous sentence
4.	SYSTEM RESPONSE	Not Identified Wrongly identified
5.	OTHER	–

evaluators were gathered to refine and enhance the approach. In the second round, the final evaluation was performed by the same evaluators.

5.2. Analysis of experiment results

The findings from the experiment are reported in this section. Table 5 displays the overall correctness of the generated test case information for each requirement set. The results show that our approach achieves a correctness of 50 % for Mdot, which correctly extracted test case information from 71 out of 142 requirements. For Pointis, the result shows that the correctness is 61.7 %, which correctly extracted 103 out of 167 requirements; and 10.3 % for Npac, which correctly extracted 3 out of 29 requirements.

The extracted test case information is further analysed, as shown in Tables 6-9. During this analysis, multiple errors that overlapped with each other were considered as a single count. The extracted information is considered incorrect if there is at least one error in any of the categories. The evaluation for the correctness is calculated by formula (1) below:

$$Correctness\% = \frac{Total\ of\ correctly\ identified\ test\ case\ information}{Total\ of\ requirements\ extracted} \quad (1)$$

The result shows that for Mdot, the correctness of actor, steps, system response and others are 78.9 %, 76.8 %, 100 % and 99.3 % respectively. The 'Other' category represents errors that are not included in the evaluation form, such as a non-functional requirement or sentences for supplementary information. For Pointis, the result shows that the correctness is 85 % for actors, 76.6 % for steps, 100 % for system response and others. The results for Npac show a correctness of 20.6 % for actors, 17.2 % for steps, 89.7 % for system response and 100 % for others.

As for the condition category, this approach will identify and analyse the conditions that are written explicitly in the requirement. Concerning that not every requirements have pre/post-conditions, the evaluation for the condition category is calculated by formula (2) below :

$$Condition\ correctness\% = \frac{Total\ of\ correctly\ identified\ conditions}{Total\ of\ conditions\ written} \quad (2)$$

The result shows that there is 7.5 % correctness for Mdot, 17.1 % for Pointis and 50 % for Npac for the condition category.

Table 10 has shown all the evaluation results in each category for each requirements dataset. The numbers include the overlapping results

Table 5
Overall result.

Requirements Specifications	Pointis	Mdot	Npac
No. of requirements without error in extracted test case information	103	71	3
Total Requirements	167	142	29
Overall correctness%	61.7	50.0	10.3

Table 6
Result by category (Pointis).

	ACTOR	STEPS	SYSTEM RESPONSE	OTHER
No. of requirements	167	167	167	167
Errors	25	39	0	0
Errors%	15.0	23.4	0.0	0.0
Correctness%	85.0	76.6	100.0	100.0

Table 7
Result by category (Mdot).

	ACTOR	STEPS	SYSTEM RESPONSE	OTHER
No. of requirements	142	142	142	142
Errors	30	33	0	1
Errors%	21.1	23.2	0.0	0.7
Correctness%	78.9	76.8	100.0	99.3

Table 8
Result by category (Npac).

	ACTOR	STEPS	SYSTEM RESPONSE	OTHER
No. of requirements	29	29	29	29
Errors	23	24	3	0
Errors%	79.3	82.8	10.3	0.0
Correctness%	20.7	17.2	89.7	100.0

Table 9
Result by conditions.

Requirements Specifications	Pointis	Mdot	Npac
(a) No. of identified conditions	10	3	6
(b) No. of identified conditions that are incorrect /ambiguous	4	0	3
(c) No. of conditions not identified	25	37	0
Errors%	82.9	92.5	50.0
Correctness% [(a-b)/$(a + c)$]	17.1	7.5	50.0

Table 10
Errors in each category.

Requirements Specifications	Pointis	Mdot	Npac
Actor			
(1a) Not Identified	2	2	0
(1b) Wrongly identified	17	16	10
(1c) Partially identified	10	12	19
Condition			
(2a) Not Identified	25	37	0
(2b) Wrongly identified	3	0	0
(2c) Identified but with ambiguous sentence	1	0	3
Step			
(3a) Not Identified	10	4	1
(3b) Wrongly identified - incorrect steps	10	6	5
(3c) Wrongly identified - missing steps	5	10	1
(3d) Wrongly identified - additional steps	13	2	1
(3e) Wrongly identified - incorrect step sequence	7	0	0
(3f) Identified but with ambiguous sentence	4	13	21
System response			
(4a) Not Identified	0	0	0
(4b) Wrongly identified	0	0	3
Other			
(5a) The requirement statement is invalid	0	1	0
Note: Overlapping not excluded			

and do not reflect the exact numbers of error used in the calculation for correctness. As shown in Table 10, The highest occurrence of errors for Mdot and Pointis is condition not identified. There are 37 out of 40 (92.5 %) conditions not identified in Mdot, with 3 requirements that have explicitly written the conditions. As for Pointis, there are 25 out of 35 (71.4 %) conditions not identified, with 10 requirements that have explicitly written the conditions. While the highest occurrence of errors for Npac is steps identified with ambiguous sentences, which is 21 out of 29 requirements (72.4 %).

It is noticeable that the overall correctness of Mdot and Pointis (positive requirements) is higher than Npac (negative requirements). However, the correctness of the condition category is higher in Npac than in Mdot and Pointis. Actor and step categories also have higher correctness in positive requirements than in negative requirements.

6. Discussion

Through the evaluation, the proposed approach has demonstrated how NLP and the unified boilerplate can be utilised to extract useful test case information from requirement specifications written in NL. In this experiment, two rounds of evaluation were carried out by two evaluators for each requirement dataset. Based on the findings from the initial round of evaluation, refinements were made to the test case information extraction process in order to increase the accuracy and improve our approach. The process of identifying actors has been improved by giving priority to the second human noun mentioned in a requirement statement. In populating the test case information, non-human nouns that represent the actor in the requirement are also taken into account and included. The experiment provides evidence that supports the feasibility of an automated test case information extraction process from requirement specifications written in NL, that can potentially be expanded to generate executable test cases when test data are available.

The proposed algorithm has also shown the ability to extract useful information from positive and negative requirements based on the boilerplate, with more than 70 % correctness in actor, steps and system response in positive requirements. However certain categories of extracted information, specifically in extracting the conditions show lower correctness. The limited scope of how the proposed algorithm identifies conditions in the requirements, which only rely on the presence of keywords (i.e., when, where, while, if), has its weakness. As any other information in other parts of requirements documentation such as UML diagram, is not considered and not analysed, the identification of test conditions is highly dependent on the keywords in the requirement itself, therefore it has to be explicitly written in the requirement. In the actor segment, the ambiguity in natural language has been a major problem for the misidentification of human actors. With the existing techniques to validate a human actor through a lexical database, the correctness is strongly influenced by the word used in the requirement statement. For example, the word “server” might be misrecognized as a human noun or an object if it is used in the requirement, e.g. ‘The streaming server will have anti-virus software to prevent infection of malicious viruses’.

On the other hand, the result of positive and negative requirements shows that the negative requirements are still a major concern in this research. It is noted that the overall correctness is 10.3 % and only condition and system response categories archive more than 50 % of correctness. The main issue identified from the analysis is due to how negative requirements are written and the way to analyse negative requirements is different compared to positive requirements, including the ambiguity from the negation word in requirement statements. This research has proposed a boilerplate that fits negative requirements (Fig. 3), however, not all requirements from the dataset used are written in the same way as the boilerplate and the requirements are much more complicated. When there is inaccurate information extracted in the process, the populated information for the test case component will be impacted. Besides, the number of negative requirements is also very limited from the available datasets, which hinders the deeper investigation.

We acknowledge that the existing information extraction falls short of providing sufficient data for achieving a comprehensive automated test process, for example the test data and test environment information was not available from the requirements. The sufficiency of information from the requirements or other sources to achieve a full test automation requires further investigation, particularly with real-world projects.

The extracted information from our approach, nevertheless, can serve as a valuable input and act as a catalyst for the test case generation process. It is able to reduce human effort, time and cost by expediting the test case generation process through the identification and population of relevant test case information such as actors, steps, and conditions automatically from requirements written in NL. Compared to other work, our approach provides more flexibility to writing the

requirements as it does not have strict rules and specific syntax. Our approach is also able to process both positive and negative requirements. However, it is also found that requirements written in a style with a higher resemblance to the boilerplate can increase the correctness of the analysis process. This insight emphasises the importance of considering writing styles and patterns in requirements. It also showcases a potential area for further investigation or optimization for both automated systems and human expertise to enhance the overall effectiveness of automated test case generation.

7. Conclusions and future work

Manual generation of test cases is laborious and extracting test case information from requirement specifications has been a challenge due to the ambiguity in NL. In this paper, an NLP approach is presented to extract useful test case information from positive and negative requirements written in English and populate the extracted information for different test case information (i.e., actor, condition, steps, system response). The proposed approach makes use of NLP techniques and language boilerplates to : (a) identify the sentence segments in the requirement, (b) extract useful test case information from requirements, and (c) populate useful test case information. The algorithm used in this approach also can identify the positive or negative statements and populate the desirable or undesirable response. The approach is evaluated by conducting an experiment using three requirement specifications and each requirement was cross-validated by two evaluators. The evaluation criteria were carefully designed to assess various aspects of the test case information, thereby providing valuable insights on the outcome of our approach. The result has presented the feasibility of the proposed approach using NLP and derived boilerplate to extract test case information from requirements written in NL.

For future work, we would like to investigate the possible ways to fully automate the test case generation process with different NLP techniques, and alternative sources for obtaining test data, optimising the test environment, and involving minimal human interventions in the testing process. We would also like to investigate the feasibility of a customizable algorithm for both positive and negative requirements, as the gap between the results of both types of requirements is notable. A boilerplate can also be explored and investigated to cater for more vaguely and ambiguously articulated requirements.

Threats to validity of results

- **Internal validity threats:** We acknowledge the possible threat to internal validity due to the evaluators' bias in the evaluation of generated test case information. To mitigate the risk, the generated outputs were distributed to one permanent evaluator and one randomly assigned evaluator. The decision by every evaluator is kept private during evaluation to prevent bias.
- **External validity threats:** Our study may face external validity concerns due to the limited selection of three requirement sets, despite choosing datasets with varied requirement writing styles. This introduces the risk of selection bias, potentially limiting the generalizability of our findings. To address this, future research should aim for a more inclusive sampling strategy, encompassing a broader range of requirement styles.
- **Construct validity threats:** A challenge arises in defining the abstract construct of correctness in our study. To counter this threat to construct validity, we've meticulously devised a detailed set of criteria to control the conceptual boundaries of our key constructs. Each criteria for the evaluation was based on the possible scenarios that can be produced on the generated results using our approach. The evaluation by the evaluators were made by validating each criteria based on the generated results. This proactive approach not only addresses concerns related to poor construct definition but also bolsters the overall credibility and reliability of our study's findings.

CRediT authorship contribution statement

Jin Wei Lim: Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis. **Thiam Kian Chiew:** Writing – review & editing, Validation, Supervision, Methodology, Funding acquisition, Conceptualization. **Moon Ting Su:** Writing – review & editing, Validation, Methodology, Formal analysis. **Simying Ong:** Writing – review & editing, Validation, Methodology. **Hema Subramaniam:** Writing – review & editing, Validation, Methodology. **Mumtaz Begum Mustafa:** Writing – review & editing. **Yin Kia Chiam:** Writing – review & editing.

Declaration of competing interest

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

Data Availability

Data will be made available on request.

Acknowledgement

This research was supported by Universiti Malaya, Malaysia under the RU Grant GPF097B-2020 - A Model-driven Approach For Generation Of Positive And Negative Requirements Based On Use Cases Written In Natural Languages.

References

Aoyama, Y., Kuroiwa, T., Kushiro, N., 2020. Test case generation algorithms and tools for specifications in natural language. In: 2020 IEEE International Conference on Consumer Electronics (ICCE). https://doi.org/10.1109/icce46568.2020.9043022.

Aoyama, Y., Kuroiwa, T., Kushiro, N., 2021. Executable test case generation from specifications written in natural language and test execution environment. In: 2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC). https://doi.org/10.1109/ccnc49032.2021.9369549.

Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F., 2014. Requirement boilerplates: transition from manually-enforced to automatically-verifiable natural language patterns. In: 2014 IEEE 4th International Workshop on Requirements Patterns (RePa). https://doi.org/10.1109/repa.2014.6894837.

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

Bird, S., Klein, E., Loper, E., 2009. Natural Language Processing With Python.

Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburn, M., 2014. NAT2TESTSCR: test case generation from natural language requirements based on SCR specifications. Sci. Comput. Program. 95, 275–297. https://doi.org/10.1016/j.scico.2014.06.007.

Ferrari, A., Giorgio Oronzo Spagnolo, & Gnesi, S. (2017). *PURE: a dataset of public requirements documents*. https://doi.org/10.1109/re.2017.29.

Fischbach, J., Junker, M., Vogelsang, A., Freudenstein, D., 2019. Automated generation of test models from semi-structured requirements. In: 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW). https://doi.org/10.1109/rew.2019.00053.

Gröpler, R., Sudhi, V., José, E., Bergmann, A., 2021. NLP-Based requirements formalization for automatic test case generation. CEUR. Workshop. Proc. 21, 18–30.

Hue, C.T.M., Dang, D.H., Binh, N.N., Truong, A.H., 2019. USLTG: test case automatic generation by transforming use cases. Internat. J. Software Eng. Know. Eng. 29 (09), 1313–1345. https://doi.org/10.1142/s0218194019500414.

IEEE Standard for Software and System Test Documentation. (2008). *IEEE Std 829-2008*, 1–150. https://doi.org/10.1109/IEEESTD.2008.4578383.

Lindsay, P.A., Kromodimoeljo, S., Strooper, P.A., Almorsy, M., 2015. Automation of test case generation from behavior tree requirements models. In: 2015 24th Australasian Software Engineering Conference. https://doi.org/10.1109/aswec.2015.23.

Loper, E., & Bird, S. (2002). NLTK. *proceedings of the acl-02 workshop on effective tools and methodologies for teaching natural language processing and computational linguistics* -. https://doi.org/10.3115/1118108.1118117.

Mai, P.X., Pastore, F., Goknil, A., Briand, L.C., 2018. A natural language programming approach for requirements-based security testing. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). https://doi.org/10.1109/issre.2018.00017.

Mavin, A., Wilkinson, P., Harwood, A., Novak, M., 2009. Easy approach to requirements syntax (EARS). In: 2009 17th IEEE International Requirements Engineering Conference. https://doi.org/10.1109/re.2009.9.

Miller, G.A., 1994. WordNet: A Lexical Database for English. ACLWeb. https://aclanthology.org/H94-1111.

Mustafa, A.M.N., Wan-Kadir, W., Ibrahim, N., Arif Shah, M., Younas, M., Khan, A., Zareei, M., Alanazi, F, 2021. Automated test case generation from requirements: a systematic literature review. Comput. Mater. Contin. 67 (2), 1819–1833. https://doi.org/10.32604/cmc.2021.014391.

Salam, M.A., Abdel-Fattah, M., Moemen, A.A., 2022. A Survey on Software Testing Automation using Machine Learning Techniques. Int. J. Comput. Appl. 183 (51), 12–19. https://doi.org/10.5120/ijca2022921919.

Shweta, Sanyal, R, 2020. Impact of passive and negative sentences in automatic generation of static UML diagram using NLP. J. Intelligent Fuzzy Syst. 1–13. https://doi.org/10.3233/jifs-179871.

Tiwari, S., Ameta, D., Banerjee, A., 2019. An Approach to identify use case scenarios from textual requirements specification. In: Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known as India Software Engineering Conference. https://doi.org/10.1145/3299771.3299774.

Tiwari, S., Shah, P., & Khare, M. (2022). *NL2RT: a tool to translate natural language text into requirements templates (RTs)*. 262–263. https://doi.org/10.1109/RE54965.2022.00035.

Wang, C., Pastore, F., Goknil, A., Briand, L., 2020. Automatic generation of acceptance test cases from use case specifications: an NLP-based approach. IEEE Transact. Software Eng. 48 (2), 585–616. https://doi.org/10.1109/tse.2020.2998503a.

Jin Wei Lim is currently a Master's by Research student at Universiti Malaya, focusing on software testing and natural language processing (NLP). His-academic journey began with a Bachelor's degree from the same institution, setting the foundation for his passion in software engineering field. Beyond academia, he was an active student engaged in various activities, demonstrating an ability to inspire and guide peers. His-commitment to academic excellence is complemented by practical experience gained as a software developer, where he is currently involved in real-world projects to enhance his technical skills.

Thiam Kian Chiew is an associate professor at the Department of Software Engineering, Faculty of Computer Science and Information Technology, Universiti Malaya, Kuala Lumpur. He heads the Research and Innovation in Software Engineering (RISE) Research Group. He was the Deputy Dean for Postgraduate Studies of the Faculty in 2016–2019. His-research areas include usability, performance and interoperability of web-based systems; as well as software engineering in e-health and m-health. Over years, he has innovated several e-health and m-Health solutions for diabetes and queue management for hospitals. In 2020, he led a team of software developers to build the COVID-19 Symptoms Monitoring System (CoSMoS) for the Universiti Malaya Medical Centre. He also involved in a few other e-Health projects for medical education and research since then.

Moon Ting Su is a Senior Lecturer in the Department of Software Engineering, Universiti Malaya. She received her Ph.D. (Computer Science) from the University of Auckland, New Zealand. She is a member of the ACM and IEEE Computer Society, and a senior member of the IEEE. Her expertise lies in software architecture and design, design patterns, object-oriented programming, end-user programming, and recommender systems. She was the treasurer for the iiWAS 2005, MoMM 2005, and Informatics 2007 international conferences. She served on the PyCon Malaysia 2015 organising committee. Before joining the education sector, she was an analyst programmer.

Simying Ong received her Bachelor's Degree in Computer Science and Doctor of Philosophy from Universiti Malaya, funded by the Malaysia Government and the University Fellowship. She is currently a Senior Lecturer in the Department of Software Engineering, Universiti Malaya. Her research interests include image/signal processing, information security (data hiding and lightweight encryption), multimedia security and their related applications.

Hema Subramaniam is a Senior Lecturer in the Department of Software Engineering at the Faculty of Computer Science and Information Technology, Universiti Malaya. She received her PhD in Software Engineering from Universiti Putra Malaysia (UPM) in 2016, following a Master of Computer Science (Software Engineering) from Universiti Selangor in 2010. A senior member of IEEE and member of IEEE Computer Society and IEEE Society on Social Implications of Technology, her research centers on software quality measurement and reusability. She actively explores mental well-being analytics via social media insights and multi-model analysis, leading software solutions for learning disabilities (LD) with grant support. Throughout her career, she has made significant contributions to software engineering, publishing numerous papers in conferences and journals. Currently, she guides Ph.D. and Master's scholars dedicated to sustainable software development, software management, and mood detection applications.

Mumtaz Begum Mustafa is an associate professor at the Department of Software Engineering, Universiti Malaya. She received a Ph.D. in Computer Science from Universiti Malaya (UM) in 2012. Her research focus is in the field of Software Engineering, Big Data Applications and Technologies, Signal Processing and Speech Technologies.

Yin Kia Chiam is currently a senior lecturer at the Faculty of Computer Science and Information Technology, University of Malaya, Malaysia. She received a Ph.D. in Computer Science and Engineering from the University of New South Wales, Australia. She has published in several reputable journals and conferences both locally and internationally. Her research focus is in the field of software engineering, data mining and healthcare analytics.