2021 IEEE 29th International Requirements Engineering Conference Workshops (REW) | 978-1-6654-1898-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/REW53955.2021.00012

Generating Sequence Diagram from Natural Language Requirements

Munima Jahan*, Zahra Shakeri Hossein Abad†, Behrouz Far*
*Department of Electrical and Software Engineering, University of Calgary, Canada, {munima.jahan, far}@ucalgary.ca
† Department of Computer Science, University of Calgary, Canada, {zshakeri}@ucalgary.ca

***Abstract*—Model-driven requirements engineering is gaining enormous popularity in recent years. Unified Modeling Language (UML) is widely used in the software industry for specifying, visualizing, constructing, and documenting the software systems artifacts. UML models are helpful tools for portraying the structure and behavior of a software system. However, generating UML models like Sequence Diagrams from requirements documents often expressed in unstructured natural language, is time consuming and tedious. In this paper, we present an automated approach towards generating behavioral models as UML sequence diagrams from textual use cases written in natural language. The approach uses different Natural Language Processing (NLP) techniques combined with some rule based decision approaches to identify problem level objects and interactions. Additionally, different quality metrics are defined to assess the validity of generated sequence diagrams in terms of expected behaviour from a given use case. The criteria we established to assess the quality of analysis sequence diagrams can be applied to similar experiments. We evaluate our approach using different case studies concerning correctness and completeness of the generated sequence diagrams using those metrics. In most situations, we attained an average accuracy factor of over 85% and average completeness of over 90%, which is encouraging.**

***Index Terms*—Sequence Diagram, Use Case Scenario, Natural Language Processing, Requirement Engineering, UML model.**

I. Introduction

Model-Driven Software Engineering (MDSE) techniques have been proven to improve software development efficiency and effectiveness through a variety of quantitative and qualitative studies [2]. With the rising complexity of software products, the way engineers design software nowadays have been changed from structured to object-oriented paradigm using Unified Modeling Language (UML) [25]. UML has become the universally-accepted language for software design and for modeling software requirements [12]. However, system requirements are usually captured in the form of Natural Language (NL) in the industry [17, 12]. Although formal models are highly efficient for designing, testing, and verification of critical systems, generating models from NL representation is complex, time-consuming and tedious. As a result, the automation of UML model generation has piqued the interest of both the scientific community and industry.

Substantial efforts have been directed towards identifying suitable models from NL requirements [6, 9] that illustrate both static and dynamic behaviors of a system. Various automated and semi-automated methods for generating UML models from NL requirements are proposed by different researchers [5, 7, 8]. However, a review of the literature in the context of automatic UML diagram generation indicates that most studies concentrate on static models like use case diagrams and class diagrams [5]. Although, behavioral models like sequence diagrams (SD) can play a significant role in use case driven object oriented (OO) software engineering, the area has not been addressed adequately, except a few [14, 29]. SDs are great tools for analyzing system behavior in the early phase of requirements elicitation to reduce the risk and cost associated with deployment. Moreover, the automation of creating SD can promote the no-code software development movement.

From the existing literature it is evident that the automation of SD generation from NL has not been explored exhaustively and provides considerable scope to work and improve. The limitations within the existing approaches include the restriction on user input. In most cases, the requirements fed to the system need to be written in a strict format [30]. Some approaches need considerable user intervention and are not fully automated [7]. To acknowledge the importance of automation in behavioural modeling and to mitigate some of the gaps in the literature, we propose an automated approach that constructs SD from use cases scenarios (UCS) written in natural language. We use a rule based approach and different NL processing techniques to identify the components of a SD and their relationship. A text script is then prepared which can be run within an open source tool to draw the actual diagram. Based on the context provided in the use case, we apply various methods to locate missing information within a sentence in order to detect system components. Our primary contribution to overcoming some of the shortcomings mentioned earlier is to provide a methodology for deriving SD from textual use cases written in free form English language with minimal user involvement. We further attempted to establish various metrics for evaluating the quality of a behavioural model. Similar experiments can benefit from the quality measures. In order to evaluate our method, we conduct a various case studies. The results from the case studies are promising and fortify the strength of the proposed approach. The overall completeness factor achieved is over 90%, while the correctness is above 85%.

The rest of this paper is structured as follows: Section II describes the related work by briefly focusing on their scope and limitations. In Section III, we provide preliminary definitions and set a stage for the methodology. The overall approach is presented in Section IV. This section also defines

978-1-6654-1898-0/21/$31.00 ©2021 IEEE
DOI 10.1109/REW53955.2021.00012

different performance metrics for evaluation. The experimental results, as well as the findings and evaluation, are presented in Section V. Finally, in Section VI, we present conclusions and future plans.

II. Related Work

According to several surveys [15], natural language is the most popular mode for representing requirements among analysts and industry practitioners. An online survey of 151 software companies found that 95% of requirements papers were represented in some form of NL [17]. Identifying concepts, relations within the requirements documents to generate visual models is important because the models enhance the clarity and understanding of the scenario. Several researches have been carried on automated and semi-automated generation of UML diagrams from NL requirements.

The efforts of extracting concepts from business specification for the purpose of conceptual modeling can be traced back to some works that have been performed in the early 1980s [23]. In the paper [23] an iterative process is presented to derive a formal specification from an informal specification written in NL. The proposed approach tries to extract the module structures from informal English description using different types of parts of speech tagging and refines the design manually in a cyclic manner. However, the earlier studies have mainly focused on the analysis of natural language requirements, and therefore heavily relied on the user intervention and manual process. The automatic extraction of object-oriented modelling has become a focus of more recent researches in this area.

Use Case Driven Development Assistant (UCDA) [27] tool helps in developing class diagrams, use-case models and also helps to visualize the models using Rational Rose tool. The tool uses syntactic analysis of the requirement statements to develop use-case diagrams. The requirements are collected from stakeholders in textual format and then analyzed using a language parser to identify the use cases and classes using a set of predefined rules. This is only an assisting tool that helps user with filling the details of the use case specification template. Moreover, it is highly dependent on a third-party tool and needs substantial user involvement. LOLITA is an NLP-based system proposed in [26] that automatically generates object model. This approach is limited to the object model generation only and cannot distinguish between classes and objects.

Linguistic assistance for Domain Analysis, LIDA [21] is a semi-automatic tool that helps designer in creating class diagrams. This tool follows the Chen's rules to associate classes with nouns, relations with verbs and attributes with adjectives. LIDA's tool can be considered as a starting point to facilitate software analysts that requires a considerable human intervention. Similar NLP techniques have been adopted in other works [11, 19] for identifying concepts in NL requirements and constructing class diagrams based on those concepts.

ABCD [3] is an automated tool for generating class diagram from natural language requirements. This approach uses the statistical and pattern recognition properties of natural language processing techniques. More than 1000 patterns are defined for the extraction of the class diagram concepts. However, the system cannot handle redundancy when synonyms are used for different objects. Also, it is unable to distinguish between association and method identification.

There are many other studies that provide automatic and semi-automatic approaches to creating UML diagrams. However, the majority of studies focus solely on creating use case or class diagrams. Only a few papers [30, 24] have focused on creating a sequence diagram, even though the sequence diagram reflects a system's behavior and is crucial for requirements analysis.

An automatic approach to derive the sequence diagram from UCSs are presented in [29]. According to the author of the publication, Abbotts's heuristics [1] is used for the object identification from NL. They represented a set of transformation rules for deriving SD from NL and provides a tool name aToucan. The major limitation of their approach is it fails to recognize domain objects and attributes. The approach also pose restrictions on the input text.

Another automated process for generating SD from use cases with a tool support is demonstrated in [28]. The authors use natural language parser to identify problem level objects and interactions between them from textual use case descriptions. This approach only deals with simple sentences and unable to handle compound and complex sentences. Grammatical knowledge pattern (GKP) is used in [25] for automated generation of behavioral models namely activity diagram and sequence diagram through lexical and syntactical analysis of requirement statements. One of the constraints of this work is that requirements documents typically contains redundancies and ambiguities, which would be reflected in any diagrams generated using this method.

As evident from the above discussion, the existing works towards semi-automated or automated creation of UML diagrams mostly concentrated on class diagrams and barely considered sequence diagrams. The methods proposed for constructing sequence diagrams from textual use cases require re-writing of the UCS following strict format or manual input. Their approach is limited to simple sentences. Our approach aims to address some of these problems. We introduce a framework for automatically producing sequence diagrams from textual use cases. Our method is able to handle compound sentences and coreferences used in the text. Furthermore, it accepts free form English text as input with very limited restrictions. The process proposed for generating sequence diagram is fully automated. Besides, we formulate various quality metrics with respect to the behavioural aspects of a SD to evaluate the accuracy and correctness of the SDs generated by our method.

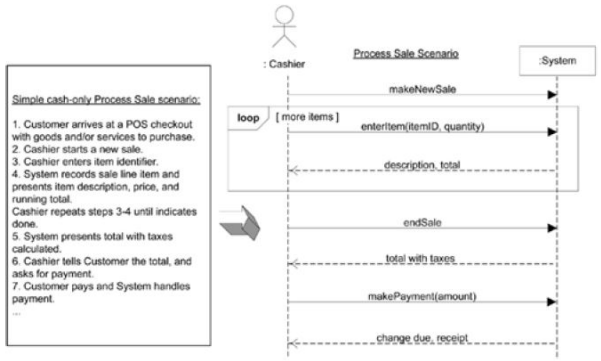

Fig. 1. Sequence diagram representing a use case for POS system [12]

III. Background and Related Terms

As part of OO design, sequence diagram is used to illustrate the collaboration between system objects and actors. A SD is a fast and easily created artifact to illustrate input and output events related to the system under consideration [12].

An example SD for a POS (Point of Sale) system including the use case text is displayed in Figure 1. This example is collected from [12].

A use case is a description of the interactions and responsibilities of a system, the "system under discussion" or "system under design", with external agents, or actors. An actor may be a person, a group of people, or a computer system. The use case text and its implied system events are input to SD creation [12].

We redefine use cases as the input to our application as a text written in simple English sentences.

A use case U is defined as a set of tuples

$$(S, C) = \{(s_1, c_1), ...(s_n, c_n)\}$$

where S represents the sentence and C represents the sentence separator, with n being the number of sentences within the use case. The sentence separator usually contains the punctuation.

Each sentence in U is then represented as a sequence of words and its POS (parts of speech) tag. A sentence S is considered as a set of tuples

$$(W, T) = \{(w_1, t_1)...(w_p, t_p)\}$$

where W represents the word and T is the POS tag for the corresponding word. The length of the sentence in terms of number of words is p.

A text usually contains different types of words. We are most interested in verbs and nouns as they represents the events and objects within a SD. More precisely, a verb $v \in W$ is a word where for the tuple (v,t) holds that t=$verb$. We denote the set of verbs with V.

A noun $n \in W$ is a word where for the tuple (n,t) holds that t=$noun$. We denote the set of nouns with N. We also use the notation $t(w)$ as the POS tag for w and $dep(w)$ as the type dependency (TD) of w. Where TD of a word is defined as the relationships between the word and the root verb in a sentence [16].

UML SDs are interaction diagrams that detail how operations are carried out. They capture the interaction between objects in the context of a collaboration. We define SD as a sequence of events represented by senders, receivers and messages collected from the use case.

A sequence diagram Sd is defined as a set of tuples

$$(E, I) = \{(e_1, 1),\ (e_2, 2),\ ...\ (e_k, k)\}$$

where, E represents the events and I is the visual orders of the events.

An event or interaction e_i collected from sentence s_i is a quadruplet

$$e_i = (p_i,\ v_i,\ m_i,\ q_i)$$

where p_i, v_i, m_i and q_i are words collected from the sentence s_i and represents the sender, verb, message and receiver respectively. The order of the event within the Sd is represented by i. We denote the set of senders, receivers, verbs and messages with P, Q, V and M respectively.

We also generalize P, Q or M as system components O. The system components are identified as the subject or an object of a sentence depending on the sentence structure. We represent the set of components as,

$$O = \{o | o \in \{P|Q|M\}\}$$

Given a UCS U the problem addressed in this paper is to find the sequence diagram Sd as a sequence of events

$$(E, I) = \{(e_1, 1), (e_2, 2).....(e_k, k)\}$$

where, E is the events such that

$$e_i = (p_i,\ v_i,\ m_i,\ q_i)$$

and I is the visual order of the event in the diagram.

IV. Proposed Method

The proposed approach is a sequential process. It starts with the preprocessing of the given textual UCS as input and ends with generating the script containing representing a SD. The input requirements are assumed to be expressed in English and adhere to the following two constraints: i) each sentence describes one single action or written as compound sentence using 'and' conjunction to specify multiple task. ii) order of the sentences within the UCS describes the flow of events sequentially. We further presume that the specifications are expressed correctly in terms of spelling and grammar. The overall workflow of the proposed approach is illustrated in Figure 2.

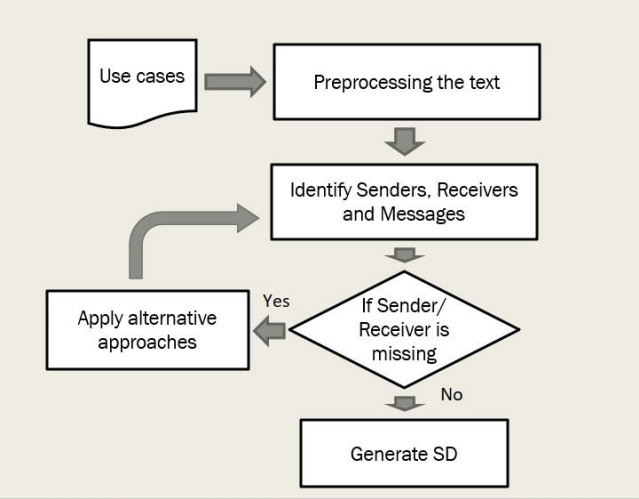

Fig. 2. Work flow of the proposed approach

In the rest of the sections, we explain and discuss the basic steps listed in Figure 2. A set of quality metrics focusing on behavioural properties of an SD are also constructed and explained in the following sections.

A. Preprocessing

The textual description of a single UCS (Use Case Scenario) is given as input to the system. In general, use cases are written in the present tense with a noun-verb-noun structure [12]. Therefore, we first filter the sentences that has a beginning pattern of Subject+Verb, and remove irrelevant sentences not specifying any action. Use cases are occasionally expressed in passive voice [22]. Keeping that in consideration our approach looks for the sentences that are written in passive voice and transforms them into active voice. Compound sentences and the use of proper nouns are also common in English prose.

In light of the foregoing, preprocessing handles a number of concerns, which can be stated as follows:

- Transforming passive voice sentences to active voice.
- Simplifying compound sentences to simple sentences.
- Solving the issues with proper nouns using the coreference resolution tools in StanFord NLP parser [16].

Compound noun phrases are also used frequently in English writing. We capture successive noun phrases as single words after preliminary cleaning of the input text. In the next section, we'll go over the procedure.

B. Collecting noun phrases

In natural language an object is often written as a noun phrase rather than a single noun. For example, "The system reads the ATM card number". The ATM card number should be considered as a single entity. To ensure that an object is represented as a collection of nouns or a combination of a modifier and nouns, the proposed approach scans all the consecutive nouns and constructs a single word as atm_card_number. It also checks any consecutive modifier and nouns for combining words as a single entity.

To avoid any conflict with upper and lower case words, the system converts the content to all lower case first, then combines the words with '_' symbol. We consider a maximum of three consecutive nouns or modifiers as a combined single word to simplify the procedure and eliminate lengthy words.

C. Identifying Sender and Receiver

Identifying the senders and receivers, as well as the messages from the requirements, is the most significant and complex process in creating sequence diagrams. We attempted to automatically determine the components of a sequence diagram from textual UCS in this research. We apply natural language processing techniques to identify the components depending on the sentence structure and the POS tagging of each word. Actors and objects are typically described by nouns. Nouns, on the other hand, can represent attributes as well. Combining the POS tag and the sentence structure helps distinguish between the two.

1) Identifying actors: At the very first step actors are identified from the given text. An actor is something with behavior, such as a person (identified by role), computer system or organization [23]. Usually a noun that acts as the subject of a sentence is a possible candidate for an actor. The system collects the subjects from each sentence as a candidate for actors and preserves it for future use.

2) Identifying senders: To identify the sender and receiver a rule-based approach is used. A set of rules are defined according to the sentence structure. After analyzing a variety of sentence structures, it was evident that the subject of a sentence is the most probable candidate for being a sender. Except, if a subject receives something from some other components. To distinguish between sender and receiver the proposed algorithm first identifies the subject of the sentence and then checks whether the subsequent verb is a synonym for "receive". If it finds a receive word the subject is identified as a receiver, otherwise it will be saved as a sender. To identify all possible receive words our approach uses the synonyms for 'receive' using WordNet [18]. The algorithm used for identifying senders, receivers and messages is displayed in Algorithm 1.

3) Identifying receivers: It is more challenging to recognize the receiver in a sentence than a sender. In general, an object of a sentence is a possible candidate for a receiver. Usually a sentence containing a preposition 'to' or 'from' has the information about a receiver. The basic rule used for detecting receiver is to look for a preposition after the verb and collect any succeeding noun phrase as the receiver. However, if the verb is a synonym of "receive", the sender and receiver is swapped (see Algorithm 1).

Some sentences may not contain any information regarding the receiver. This issue is not mentioned or addressed in any existing literature. We have applied different approaches to mitigate the limitation. For example, "System sends the price list" does not implicitly clarify to whom the system is sending the price list. However, if the previous sentences demonstrate that the system was interacting with customer, it

Algorithm 1: Find `Events` for a `SD` from a given `UCS`

FindEvents (U, Sd)
- **inputs:** A textual UCS U =$(s_1, s_2, ...s_n)$, A set of words synonyms to 'receive' RW
- **output:** A sequence of event as Sd=$(e_1, e_2, ...e_n)$
- **foreach** *sentence* $s_i \in U$ **do**
  - **foreach** *word* $w_j \in s_i$ **do**
    - **if** $dep(w_j) = nsubj \wedge dep(w_{j+1}) = ROOT$ **then**
      - $p_i \leftarrow w_j$;
      - $v_i \leftarrow w_{j+1}$;
      - Find next w_k such that $t(w_k) \in \{NN, NNP\} \vee$ $dep(\mathrm{w}_k)$=obj
      - $m \leftarrow w_k$;
      - Find next word w_l such that $t(w_l) = IN$
      - $q_i \leftarrow w_l$
    - **end**
    - **if** $v_i \in RW$ **then**
      - $swap(p_i, q_i)$;
    - **end**
  - **end**
  - $e_i \leftarrow createEvent(p_i, v_i, m_i, q_i, i)$;
  - add(e_i,Sd); //add the event as part of the SD Sd
- **end**
- **return** Sd;

Algorithm 2: Handle missing `Receiver` in a given `SD`

HandleMissingR (Sd)
- **inputs:** A sequence of event as Sd=$(e_1, e_2, ...e_n)$ A list of words consider as $Single\ words\ SW$
- **output:** Updated sequence diagram $Sd^{'}$=($e_1^{'}$,$e_2^{'}$,...$e_n^{'}$)
- **foreach** *event* $e_i \in Sd$ **do**
  - **if** $q_i = NULL$ **then**
    - **if** $v_i \in SW$ **then**
      - $q_i \leftarrow p_i$
    - **end**
    - **else**
      - **if** i=*1* **then**
        - Find next event e_k such that $p_i \neq p_k \wedge k > i$;
        - $q_i \leftarrow p_k$;
      - **end**
      - **else**
        - Find previous event e_l such that $p_i \neq p_l \wedge l < i$;
        - $q_i \leftarrow p_l$;
      - **end**
    - **end**
  - **end**
- **end**
- $e_i^{'} \leftarrow createEvent(\mathrm{p}_i, v_i, m_i, q_i)$;
- $Sd^{'} \leftarrow update(\mathrm{e}_i, e_i^{'}, Sd)$; //update e_i with $e_i^{'}$ in SD Sd
- **return** $Sd^{'}$;

is most probably the case that the system sends a price list to the customer. To identify the potential receiver we use a few different steps. If the very first sentence in the UCS does not have a receiver details, then the next sender in the list that is different from the current sender is selected as receiver. Otherwise, the first sender within the previous event list that is different from the sender in the current event is selected (see Algorithm 2).

There could be another possible situation where the receiver is not mentioned directly, such as: "The system updates the price list". A system component can perform some actions by itself and may not need to send anything to anyone. Especially when it updates something, or processes something. Acknowledging these situations a list of words are classified as 'Single words' where the sender itself is a receiver. To make the word list more comprehensive, all possible synonyms for the selected single words are added using WordNet [18]. Single words include words like 'process,' 'handle,' 'update,"manage,' and so on.

The process followed for handling missing object is demonstrated in Algorithm 2.

D. Illustrative Example

This section demonstrates the proposed approach with an example. We implemented our methodology as an executable program using Python 3.6 and the StanFord CoreNLP API. We demonstrate the example of NextGen POS system collected from [12]. The use case is described as follows:

Cashier starts a new sale. Cashier enters item identifier. System updates the sale line item. System presents item description and price. Price is calculated from a set of price rules. System presents total with taxes calculated. Cashier tells customer the total. Cashier asks for payment. Customer pays. System handles payment. System logs the completed sale. System sends sale information to the external Accounting system. System sends sale information to the Inventory system. System presents receipt. Customer leaves with receipt and goods.

The use case is read as a text input and preprocessed according to the steps described earlier. For example, the sentence "System presents item description and price" is broken into two separate sentences "System presents item description" and "System presents item price". The preprocessing also resolves any issues with coreferences. After the preliminary processing is done the text is annoted by the Stanford NL parser and noun phrases are collected as mentioned in previous section. The system components (sender, receiver, messages) are then identified according to the Algorithm 1.

The components identified primarily from the example UCS are presented in Table I. From Table I it is clearly visible that most of the interaction is missing the receiver information. To complete the missing receivers we follow the procedure in Algorithm 2.

One of the most challenging tasks is identifying the sender or receiver when there is no implicit information in the sentence. Following the steps in Algorithm 2, the receiver for the first sentence is identified as the 'system'. Because, the next sender in the table other than 'cashier' is 'system'. Similarly, the sentence, 'system presents receipt' does not clearly mention who is the receiver. Therefore the object 'cashier' is selected as the receiver.

We get the revised Table II after finding all missing components.

If some sender or receiver information remain empty after applying all possible techniques for missing information, it is marked as unknown as shown in Figure 4.

E. Generating sequence diagram

In this step, the information collected in Table II is used to generate a text script compatible with plantUML [10]. The generated text script is then exported in a file and copied to the associated online tool for drawing SD using plantUML [10]. The sequence diagram (SD) generated from the given example using plantUML is presented in Figure 3.

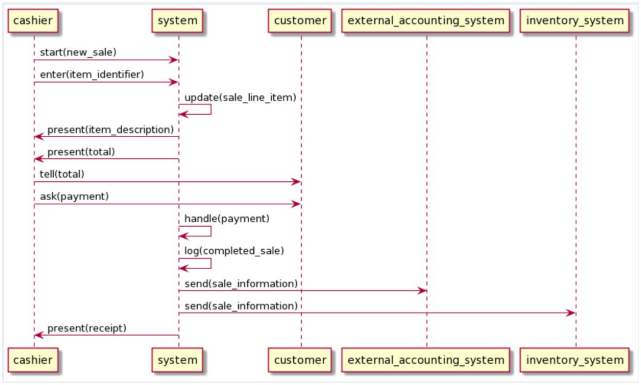

Fig. 3. Sequence diagram generated from the information in Table II using plantUML.

The reason for using plantUML is two fold. First, plantUML is a component that allows to quickly write different diagrams as text commands an provides easily accessible online tool to draw the diagrams from the text. Second, as we have all the necessary information for generating the SD as given in Table II, different types of script can be produced if necessary. Moreover, generating actual image for the diagram is considered as out of scope for this research.

F. Quality Measures

Evaluating sequence diagram in terms of correctness and completeness is always disputable and may subject to the individual's expertise and understandings. Comparing the results with other work can be critical too, as the input structure and the amount of human intervention are different for distinct approaches. We put our effort to define a set of measures for assessing the validity and completeness of the generated SDs. We will discuss the measures in following sections.

1) Correctness: A SD represents the system behaviour for a single use case scenario as a series of interactions between system elements. Interactions, also known as events, are made up of four key components: a message, a sender, a receiver, and the visual order of the interaction within the scenario. Therefore, identifying all components correctly is a major concern regarding the quality of the diagram. However, it is also possible that some components have more critical role within that specific event compared to others. We therefore, construct a quality measure that integrates individual variables with a given weight, taking all of these aspects into account as contributing factors. The correctness of an interaction e_i is defined as:

$$e_{i_{corr}} = \frac{w(m) + w(p) + w(q) + w(i)}{3W_{ei}}$$

where, $w(m)$, $w(p)$, $w(q)$, and $w(i)$ represents the weight value for each element message, sender, receiver and the visual order respectively depending one whether the particular element is identified correctly or not. W_{ei} represents the combined weight for all components. The weights can be defined as a function of individual elements and have different values based on whether the system detect the element correctly. We define our weights as follows:

$$w(x) = \begin{cases} 0 & x\ identified\ incorrectly \\ 1 & x\ identified\ correctly \\ where,\ x \in \{m,p,q,i\} \end{cases}$$

Also,

$$W_{ei} = \sum_{x \in \{m,p,q,i\}} w(x), for\ x\ identified\ correctly$$

While considering the behavior depicted within a scenario, some interaction may have more impact on the system than others. Keeping the priority in the context we assign different weights for individual events which can be distributed among the elements of the event as well. Allowing precedence among events we define the total weighted correctness of a SD Sd as:

$$F_{corr}(Sd) = \frac{1}{W_T} \sum_{i=1}^{n} w_i e_{i_{corr}}$$

where, n is the total number of interactions within the SD Sd and $e_{i_{corr}}$ is the correctness for event e_i. The total weight W_T is defined as:

$$W_T = \sum_{i=1}^{n} w_i$$

The cumulative performance factor in terms of correctness for the overall approach is calculated as follows:

$$\mu_{corr} = \frac{1}{T} \sum_{j=1}^{T} F_{corr}(Sd_j)$$

TABLE I
IDENTIFIED COMPONENTS FOR SD.

SNo	Verb	Sender	Receiver	Message	Sentence
0	start	cashier		new_sale	cashier starts a new_sale.
1	enter	cashier		item_identifier	cashier enters item_identifier.
2	update	system		sale_line_item	system updates the sale_line_item.
3	present	system		item_description	system presents item_description.
4	calculate			set	price is calculated from a set of price_rule.
5	present	system		total	system presents total with taxes calculated.
7	tell	cashier	customer	total	cashier tells customer the total.
8	ask	cashier		payment	cashier asks for payment.
9	handle	system		payment	system handles payment.
10	log	system		completed_sale	system logs completed_sale.
11	send	system	external_accounting_system	sale_information	system sends sale_information to the external_accounting_system.
12	send	system	inventory_system	sale_information	system sends sale_information to the inventory_system.
13	present	system		receipt	system presents receipt.
14	customer_leave			receipt	customer leaves with receipt and goods.

TABLE II
UPDATED TABLE WITH COMPLETE INFORMATION

SNo	Verb	Sender	Receiver	Message	Sentence
0	start	cashier	system	new_sale	cashier starts a new_sale.
1	enter	cashier	system	item_identifier	cashier enters item_identifier.
2	update	system	system	sale_line_item	system updates the sale_line_item.
3	present	system	cashier	item_description	system presents item_description.
4	present	system	cashier	total	system presents total with taxes calculated.
5	tell	cashier	customer	total	cashier tells customer the total.
6	ask	cashier	customer	payment	cashier asks for payment.
7	handle	system	system	payment	system handles payment.
8	log	system	system	completed_sale	system logs completed_ sale.
9	send	system	external_accounting_system	sale_information	system sends sale_information to the external_accounting_system.
10	send	system	inventory_system	sale_information	system sends sale_information to the inventory_system.
11	present	system	cashier	receipt	system presents receipt.

where, T is the total number of SDs and Sd_j represents the jth sequence diagram considered.

2) Completeness: We also asses the completeness of the sequence diagrams reformulating the quality measures proposed in [30]. The completeness of a SD signifies how many components and interactions are identified correctly with respect to the referenced diagrams.

The completeness of a SD is measured as the ratio between the total components correctly generated from the program to the number of components found in the reference SD.

The completeness factor of a SD Sd with respect to the component O is defined as:

$$F_{com}(Sd(O)) = 1 - \frac{\#\ of\ O\ detected\ incorrectly}{\#\ of\ O\ in\ the\ Referenced\ SD}$$

where O can be either the set senders (P) or receivers (Q) or messages (M) or objects (O).

The completeness of each SD is calculated with respect to sender, receiver and messages separately and then an average completeness factor is measured for each sequence diagram.

The average completeness of a SD Sd is defined as:

$$Avg_{com}(Sd) = \frac{1}{3} \sum_{o \in \{P,Q,M\}} F_{com}(Sd(o))$$

The cumulative completeness of the system is defined as:

$$\mu_{com} = \frac{1}{T} \sum_{j=1}^{T} Avg_{com}(Sd_j)$$

where, T is the total number of SDs under consideration.

3) Similarity Measure: The acceptance of a SD also depends on how accurately it models the use case in terms of the behavior described by a UCS. We define a similarity measures between the given use case and the generated SD

to verify the completeness of the diagram with respect to the given scenario. Each sentence in requirements represents an interaction or event. The verbs are usually considered as the actions between objects mentioned as noun in natural language. In addition, subject of any sentence is considered as a strong candidate for being an actor. To examine the similarity between each sentence and the corresponding interaction we create two separate vectors: i) a use case vector that contains all nouns and verb in the sentence including the subject; ii) an interaction vector containing sender, receiver and message from the corresponding interaction.

We define the use case vector as:

The use case vector $V_{uc}(i)$ for the ith sentence s_i is

$$V_{uc}(i) = \{w|t(w) \in \{VBZ \vee \ NOUN\} \wedge w \in s_i\}$$

The interaction vector $V_{in}(i)$ for ith interaction in the diagram is defined as:

$$V_{in}(i) = \{o|o \in e_i \wedge o \in \{P|Q|M\}\}$$

The similarity λ between the two vectors is then calculated using Cosine Similarity measures [13].

$$\lambda = CosineSimilarity(V_{uc}(i), V_{in}(i))$$

The overall similarity between the UCS and the SD is determined as:

$$\lambda_{avg}(Sd) = \frac{1}{n}\sum_{i=1}^{n} CosineSimilarity(V_{uc}(i), V_{in}(i))$$

where, n is the total number of events.

V. Performance Evaluation

Evaluation can be either methodological or experimental. As reported in the related work section, only a few approaches exist that are fully automated and generate sequence diagrams from textual user requirements similar to our approach. That is why, we concentrate our efforts on methodological evaluation of the generated sequence diagrams in terms of behavioural characteristics.

A. Case Study

To evaluate the proposed approach based on the measures defined in preceding sections we collect three case studies from different domain used in some existing literature. The main goal for the case study is to demonstrate the efficiency of our approach in generating SD and to establish the applicability of the method in various domains. The three case studies are NextGen POS (Point of Sale), ATM (Automated Teller Machine) and ARENA (tournament management system). The NextGen POS is collected from [12] and used in the illustrative example. The UCS for "ProcessSale" that is written in ten sentences is used as the first case study. The UCS for ATM "WithdrawFund" is defined in [29] as a text of length twelve. Similarly, ARENA comes from [4], where a use case called "Announce Tournament" is described by six sentences. The reason behind choosing these case studies is because all the above system models come from well-respected, expert sources, well-known book authors and provides with sequence diagrams which can be used as a reference for our evaluation process.

TABLE III
Correctness factor calculated for different case studies

Case Study	# of events	$F_{corr}(Sd)$ %	μ_{corr} (%)
NextGen	13	100	
ATM	11	81	88.7
ARENA	7	85	

Some of the evaluation results are presented in the following sections. The correctness factors calculated for different case studies are given in Table III. The completeness factor prepared for different case studies are presented in Table IV.

The cumulative average completeness and correctness factors calculated are 88.7% and 91% respectively as shown in Table III and Table IV.

The overall similarity measures between the given UCS and generated SD for the case studies are illustrated in Table V.

B. Discussion

From the case studies we can observe that our approach fits in different application domains. The completeness and correctness factors also demonstrates the strength of our approach. It is noticeable from the NextGen POS example that the diagram collected from [12] only represents the high level system sequence diagram, whereas our approach generates more detailed SD. The overall completeness and correctness factor achieved from the case studies are also encouraging.

Compared to [28] and [29] our approach can handle compound sentences and coreference without any human intervention. The restriction rules are very limited compared to others. We also provide a similarity measure between the given scenario and the generated SD which reflects the correctness of the diagram with respect to the UCS. This factor can be used as a self evaluation tool for individual SD when there is no reference SD to compared with.

However, our system may fail to identify all components in certain scenarios like the one shown in Figure 4.

If we scrutinize the sentences closely we can see that for the first two sentences the receiver is not mentioned explicitly. However, as a software engineer we can easily speculate the possible receiver as 'System' which is not automatically identifiable from any of the sentences within the use case. Therefore, the receivers are identified as unknown. At this stage, our technique is also incapable of dealing with loops and branching.

C. Threat to the Validity

Our approach uses POS tag and TDs to identify different components of a SD and highly depends on the Stanford parser for the task, which can be considered as a threat to the validity of our approach. The parser generates POS tags with accuracy about 97% [28] and type dependency with accuracy 84.2% [16]. However, any minor grammatical error

TABLE IV
EVALUATION RESULTS FOR COMPLETENESS

Case Study	$F_{com}(S)$ %	$F_{com}(R)$ %	$F_{com}(M)$ %	$Avg_{com}(Sd)$ %	μ_{com} (%)
NextGen	100	90	100	96	
ATM	82	82	100	88	91
ARENA	85	85	100	90	

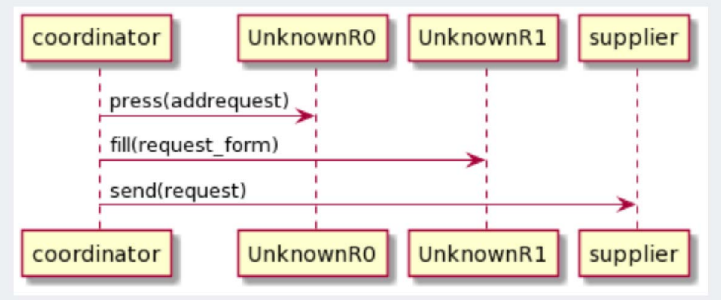

Fig. 4. SD generated with multiple unknown object or receiver.

TABLE V
SIMILARITY MEASURE BETWEEN UCS AND CORRESPONDING SD

Case Study	# of events	$\lambda_{avg}(Sd)$ %
NextGen	13	85
ATM	11	80
ARENA	7	85

can identify the POS tag incorrectly which eventually affects the performance of our algorithm. We assume, though, that the specifications are produced by competent software engineers using standard templates, and therefore the likelihood of errors is low. Furthermore, there are a variety of sophisticated text editors and tools [20] that can check for spelling, grammar, punctuation, clarity, engagement, and delivery errors, which can be used to minimize the writing flaws.

Another concern that may arise questioning the validity of our approach is the reference SDs used for the evaluation of our proposed method. As these SDs are compiled from the existing literature, the quality of the SDs may affect the evaluation results. However, the SDs are collected from well-respected expert sources and well-known books to strengthen the confidence in the diagrams' quality.

VI. CONCLUSION AND FUTURE WORK

Generating sequence diagrams from textual requirements is a complex and time-consuming task, and automating it is considerably more challenging. However, it is essential to analyze the behavior of a system to ensure the safety and security of its users. Sequence diagram is one of the most popular and useful tools to model system behaviour and to verify system requirements.

In this paper, we offer a method for automatically generating SD from English-language use case situations. We investigated the approach using different case studies from diverse fields and found promising results in terms of completeness and correctness of the generated SDs. However, this report only offers the preliminary findings of our ongoing research, and much more work remains to be done. In order to accommodate more complex sentences, loops and branches in our designs we are experimenting with different sentence patterns and structures. To improve the outcomes, we are also looking into different options for coping with missing data, such as Granger causality and conditional probability.

The system reported in this work is implemented in Python 3.6. However, there is no user interface or publicly accessible tool. Our long-term goal is to provide a software tool that anyone can use to generate SD from textual requirements.

ACKNOWLEDGMENT

This research is partially supported by NSERC (Natural Sciences and Engineering Research Council), Canada and AITF (Alberta Innovates Technology Futures).

REFERENCES

[1] Russell J Abbott. Program design by informal english descriptions. *Communications of the ACM*, 26(11):882–894, 1983.

[2] Roberto Acerbis, Aldo Bongio, Marco Brambilla, Massimo Tisi, Stefano Ceri, and Emanuele Tosetti. Developing ebusiness solutions with a model driven approach: The case of acer emea. pages 539–544, 07 2007.

[3] Wahiba Ben Abdessalem Karaa, Zeineb Ben Azzouz, Aarti Singh, Nilanjan Dey, Amira S. Ashour, and Henda Ben Ghazala. Automatic builder of class diagram (abcd): an application of uml generation from functional requirements. *Software: Practice and Experience*, 46(11):1443–1458, 2016.

[4] Bernd Bruegge and Allen H Dutoit. Object–oriented software engineering. using uml, patterns, and java. *Learning*, 5(6):7, 2009.

[5] Omer Salih Dawood et al. From requirements engineering to uml using natural language processing–survey study. *European Journal of Engineering and Technology Research*, 2(1):44–50, 2017.

[6] Deva Kumar Deeptimahanti and Muhammad Ali Babar. An automated tool for generating uml models from natural language requirements. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 680–682. IEEE, 2009.

[7] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, pages 165–174, 2011.

[8] Sarita Gulia and Tanupriya Choudhury. An efficient automated design to generate uml diagram from natural language specifications. In *2016 6th international conference-cloud system and big data engineering (Confluence)*, pages 641–648. IEEE, 2016.

[9] HM Harmain and Robert Gaizauskas. Cm-builder: A natural language-based case tool for object-oriented analysis. *Automated Software Engineering*, 10(2):157–181, 2003.

[10] Kwai Hing Heung. A tool for generating uml diagram from source code. 2013.

[11] Mohd Ibrahim and Rodina Ahmad. Class diagram extraction from textual requirements using natural language processing (nlp) techniques. In *2010 Second International Conference on Computer Research and Development*, pages 200–204. IEEE, 2010.

[12] Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012.

[13] Baoli Li and Liping Han. Distance weighted cosine similarity measure for text classification. In *International conference on intelligent data engineering and automated learning*, pages 611–618. Springer, 2013.

[14] Liwu Li. A semi-automatic approach to translating use cases to sequence diagrams. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No. PR00275)*, pages 184–193. IEEE, 1999.

[15] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, 2004.

[16] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[17] Luisa Mich, Mariangela Franch, and Pier Luigi Novi Inverardi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(2):151–151, 2004.

[18] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[19] Priyanka More and Rashmi Phalnikar. Generating uml diagrams from natural language specifications. *International Journal of Applied Information Systems*, 1(8):19–23, 2012.

[20] Ruth ONeill and Alex Russell. Stop! grammar time: University students' perceptions of the automated feedback program grammarly. *Australasian Journal of Educational Technology*, 35(1), 2019.

[21] Scott P Overmyer, L Benoit, and R Owen. Conceptual modeling through linguistic analysis using lida. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 401–410. IEEE, 2001.

[22] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML*. Springer, 1999.

[23] Motoshi Saeki, Hisayuki Horai, and Hajime Enomoto. Software development process from natural language specification. In *Proceedings of the 11th international conference on Software engineering*, pages 64–73, 1989.

[24] Laura Mendez Segundo, Rodolfo Romero Herrera, and K Yeni Perez Herrera. Uml sequence diagram generator system from use case description using natural language. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA 2007)*, pages 360–363. IEEE, 2007.

[25] Richa Sharma, Sarita Gulia, and KK Biswas. Automated generation of activity and sequence diagrams from natural language requirements. In *2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 1–9. IEEE, 2014.

[26] Mark H Smith, Roberto Garigliano, and Richard G Morgan. Generation in the lolita system: An engineering approach. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, 1994.

[27] Kalaivani Subramaniam, Dong Liu, Behrouz Homayoun Far, and Armin Eberlein. Ucda: Use case driven development assistant tool for class model generation. In *SEKE*, volume 7804, page 16th. Citeseer, 2004.

[28] Jitendra Singh Thakur and Atul Gupta. Automatic generation of sequence diagram from use case specification. In *Proceedings of the 7th India Software Engineering Conference*, pages 1–6, 2014.

[29] Tao Yue, Lionel C Briand, and Yvan Labiche. An automated approach to transform use cases into activity diagrams. In *European Conference on Modelling Foundations and Applications*, pages 337–353. Springer, 2010.

[30] Tao Yue, Lionel C Briand, and Yvan Labiche. atoucan: an automated framework to derive uml analysis models from use case models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):1–52, 2015.