Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 07, No. 02, June 2021.

AUTOMATIC GENERATION OF UML DIAGRAMS FROM SCENARIO-BASED USER REQUIREMENTS

Abdelkareem M. Alashqar

(Received: 18-Mar.-2021, Revised: 9-May-2021 and 19-May-2021, Accepted: 22-May-2021)

ABSTRACT

Effective software modeling tools are necessary for successful achievement of software engineering activities, especially when working in the analysis and design phase. Automating these tools facilitates work, makes it more productive and reduces cost and time of development. This paper aims at the development and validation of a method and a software tool for automatic generation of UML diagrams when following the approach of object-oriented development. These diagrams are generated from scenario-based requirements in order to facilitate the modeling process. So, a template of scenario-based requirements and its components are identified and constructed. Then a method including an algorithm is designed and implemented based on natural language processing (NLP) to generate UML diagrams automatically from the scenario-based requirements. The diagrams include sequence and class diagrams. The ability, performance and benefits of the proposed method and the software tool are reported by experimental results.

KEYWORDS

Software engineering, Object-oriented, UML, Use cases, Scenarios, Natural language processing.

1. INTRODUCTION

Currently, object-oriented (OO) approaches are popular in developing software systems. The Unified Modeling Language (UML) is a powerful notation that is utilized in the analysis and design of OO software development process. Traditionally, software developers perform a lot of work in understanding the documented software requirements in order to build the required UML diagrams before drawing them using Computer Aided Software Engineering (CASE) tools. However, additional work is also needed by software developers for specific understanding and gaining familiarity and experience of these CASE tools. Because software requirements are typically written in natural languages such as English, Natural Language Processing (NLP) tools can be utilized for automatic generation of software models, such as UML diagrams.

The authors in [1] argue that a complete automation of constructing UML diagrams using NLP appears to be impossible and hence more advanced automation is needed to be achieved. Software requirements in the OO development process are widely elicited and analyzed using UML use cases 0. Software functional requirements are described in terms of use cases and actors [3] and written in a restrictive way using system scenarios. A scenario represents a particular path in one of the use cases' set that constitutes the main use case diagram of the software being developed. Scenarios have a powerful method in sharing the needs of stakeholders and in describing the story of functional behavior [2]. UML sequence diagram is normally linked to the realization of the use case of the system being developed. It shows the objects involved in the functionality of the scenario and their interactions organized in time sequence in addition to the messages exchanged between these objects [4].

One way for depicting the class diagram is to capture objects involved in the sequence diagram and make aggregations' relationships based on the interactions among objects. Sequence diagram always helps in depicting the interaction (dynamic) view, while class diagram always helps in depicting the structural (static) view of the software. However, there is a lack of work in automatic generation of dynamic views of the software being developed, because most of work goes to automatically building a static view of the software such as its structure. Sequence diagram helps software developers verify whether the identified classes are sufficient for achieving user requirements written in scenario format or not [5].

This paper introduces a method that includes a designed algorithm based on NLP for analyzing and

A. M. Alashqar is with Faculty of Information Technology, The Islamic University of Gaza, Palestine. Email: `aashgar@yahoo.com`

extracting information from scenario-based user requirements written in English text. The method will be developed as a software tool for automatic generation of UML (AGUML) diagrams, especially sequence diagrams and class diagrams.

The rest of this paper is organized as follows: Section 2 presents related work. Section 3 introduces the research methodology where the algorithm of analyzing scenario-based requirements is designed. The main structure of our software tool for automatic generation of UML diagrams (AGUML) is described in Section 4. Section 5 presents experiments performed by AGUML, in addition to the analysis of results. Section 6 concludes the paper and provides a future work outlook.

2. Related Work

In this section, we provide a literature review for some published research works in the field of generating UML diagrams automatically and how NLP can help in achieving this automation. Specifically, we focus on the research works that tackle the problem of generating sequence and class diagrams from written user requirements.

The UML is a powerful notation that is used particularly in OO analysis and design activities when developing software systems. However, it is considered a semi-formal language that needs semantics when constructing and verifying its different types of diagrams [6].

The notion of using NLP in OO modeling has been initially proposed by [7][8][9], where the authors mentioned that the object and its properties can be recognized by searching nouns, while the object's operations and the interactions among objects can be recognized by searching verbs in written software requirements.

There have been many works that adopt these founded concepts in building CASE tools for automatic generation of UML diagrams from informal natural language. Most of these tools focus on generating class diagrams, such as REBUILDER UML [10], CM-Builder [11], LIDA [12], MOVA [13], UMLG [14], RACE [15], DC-Builder [16] and RAPID [17]. The authors in [18] developed a tool named GOOAL based on a semi-formal language (4WL) for generating class models automatically from semi-structured natural language. Their tool also can construct initial and general sequence diagrams.

For other types of UML diagrams, the authors in [19] developed a plugin for automatic transformation of user stories into UML use case diagrams depending on the basis of NLP techniques. Whereas, the authors in [20] proposed a semi-automated approach for generating use case diagrams from user requirements that are mainly written in Arabic.

Recently, the authors in [21] developed an algorithm for reading and processing user stories stored in a text file, then generating an Extensible Markup Language (XMI) file for each individual user story and finally transforming the XMI file into use case diagram. The authors in [22] proposed a semi-automated approach for generating sequence diagrams from user requirements written in Arabic. The generated sequence diagrams are expressed using XMI format in order to be drawn using specialized software drawing tools.

In this paper, a method that includes an algorithm and a software tool for generating UML sequence and class diagrams is proposed. The differences between our work and the previously stated related published works is that our proposed method utilizes the software requirements that are written as user scenarios. User scenarios are considered a powerful technique for writing detailed software descriptions when adopting use case approach in the process of requirements' elicitation and analysis when developing software. To our knowledge, no such work for automatic generation of sequence diagrams from user scenarios is achieved. Moreover, our work introduces a way for constructing class diagrams from the generated sequence diagrams by understanding the relationships and interactions among the participating objects. Moreover, our developed tool has a complete capability for generating UML disarms in order to be used in user documentation.

3. Research Methodology

The scenario-based requirements template and its components are defined based on a theoretical perspective and previous work. An algorithm is designed and implemented as a CASE tool named AGUML in order to process the scenario text using NLP and semantics. Then, the accuracy of the algorithm is presented with the analysis of performed experimental results.

3.1 Use Case and Scenarios

Scenarios are a powerful solution to the complexity of software development. They are used as telling stories to help system stakeholders share a complete view about the software being developed and to help in avoiding any missing problems [23]. The scenario is defined as: "one sequence of events that is one possible pathway through a use case" [24]. Therefore, a use case may include one or more scenarios. A scenario can be used as a tool for determining and validating software requirements. There is a variety of scenario descriptions used in the literature. However, the description of a scenario should include at least the following parts [25]-[26]:

- Initial Assumption: This part includes a description of software users' expectations at the start point of the scenario.
- Basic Path: This part depicts the normal sequence flow of events as expected by software users. It may also be called "Sunny Day Scenario" or "Happy Path".
- Alternate Path: This part includes the alternative set of steps that may be produced as a result of testing events of the basic path. These steps always run in parallel with the basic path. It may also be called "Rainy Day Scenario" or "Unhappy Path".
- Exception Path: This part includes output results for unsuccessful steps in the basic path.
- System State on Completion: This part includes the description of the system state at the end point of the scenario.

The author in [27] observed that developers often think of application through a relatively small number of typical interactions with users, such as the following:

1) User does …
2) Application does …
3) Application does …
4) User does …
5) Application does …
6) etc.

And these are called use cases, which are often used in conjunction with OO analysis and design methods. Use cases can be used as a starting point of requirements' analysis in order to derive classes from them [27]. A use case is always identified by its name, the actor which is the user type of the software and the interaction between the actor and the software. For instance, "Add new student" can be a typical use case for the course registration system with the "Registrar" as an actor. This use case may include the following sequence of steps:

1) Application shows an options' screen.
2) Registrar selects addition from options' screen.
3) The system presents a student entry screen.
4) Registrar enters student data.
5) Registrar submits student data.
6) System saves student data into database.

As seen from the above scenario example, each step in the scenario represents an English sentence of the form "Subject and Predicate".

3.2 Algorithm Design

We have designed an NLP algorithm that has a capability of analyzing scenario-based requirements written in restrictive English text to extract objects, their interactions and the messages passed among them in order to draw UML sequence and class diagrams. The following rules should be met before applying the NLP algorithm:

- The scenario is written as a sequence of English language sentences, where the structure of the scenario represents a typical normal flow of interactions between the user and the software, as stated in the previous subsection.
- Each sentence is separated from the next one by the normal end of line.
- Each sentence represents an action occurring in the software. This action can usually be initiated

externally by the user or internally by an object inside the software and appears in the active voice format.

The pseudocode of the proposed NLP algorithm is shown in Figure 1 and the main steps are described as follows:

1) At the beginning, the algorithm reads the scenario text, then it normalizes it (line 2). The normalization process produces a uniform word for some other different words that are equivalent. For example, replace any word in the set: "application", "applications", "system", "systems", "software" and "package" with a uniform word: "application". This is because these different words are considered equivalent based on user intention of software requirements.

Then, the algorithm declares and initializes some buffers to be used in storing important information during scenario processing (line 3), where, "umlSequenceScript" and "umlClassScript" buffers are used to store a specialized script as text for the whole scenario to be utilized later in drawing the needed UML diagrams. The "actor" buffer is used to store the name of actor as text if it exists. As a rule, there is only one actor for each scenario.

```
 1. Read the scenario text
 2. Normalize the scenario text
 3. Initialize and Set umlSequenceScript = "", umlClassScript = "",  actor = ""
 4. FOR each sentence in the normalized scenario text
 5. Produce a list of tokens
 6.     Produce a parsed tree (pTree) from tokens
 7.     If (pTree) has S tree (sTree)
 8.         and (sTree) has NP (npTree) and VP (vpTree) subtrees respectively
 9.         and (vpTree) has at least one NP subtree
10.             Initialize and Set parts[] = {"", "", "", ""}
11.             Get all words that starts with NN label from (npTree)
12.                 Then concatenate them and set them to parts[0]
13.             Get tagged word that labeled with NNP from (npTree) and Set it to actor
14.             Get word labeled starts with VB from (vpTree) and set it to parts[2]
15.             If (vpTree) has only NP labeled tree
16.                 Get all words that starts with NN label from (vpTree)
17.                     Then concatenate them and set them to parts[1]
18.             END IF
19.             If (vpTree) has two NP labeled trees
20.                 Get all words that starts with NN label from the first child of (vpTree)
21.                     Then concatenate them and set them to parts[3]
22.                 Get all words that starts with NN label from the second child of (vpTree)
23.                     Then concatenate them and set them to parts[1]
24.             END IF
25.     END IF
26.     Construct uml script for sequence from parts[] and add it to umlSequenceScript
27.     Construct uml script for class from parts[] and add it to umlClassScript
28. END FOR
29. IF Actor is not empty
30.     Add it as a prefix to umlSequenceScript
31. END IF
```

Figure 1. Algorithm pseudo-code for automatic generation of UML diagrams.

2) The algorithm then splits the scenario into sentences and for each individual sentence repeats important processing steps (lines 4-28) as follows:

The loop starts by tokenizing the sentence. For example, the output of the scenario sentence: "The user logs into the system" will be tokenized into: [The] [user] [logs] [into] [the] [system]. Then, morphological analysis is applied on each tokenized statement to identify the different parts of speech (POS). In POS process, each word in the sentence is tagged or labeled individually such as: (The/DT) [user/NN] [logs/VBZ] [into/IN] [the/DT] [system/NN], where "DT" means determiner, "NN" means noun, singular or mass, "VBZ" means verb or third person singular present and "IN" means preposition or subordinating conjunction. Then, for semantic analysis, a syntactical and lexical analysis is performed to produce a parsed tree form such as the following which is a typical output from Stanford CoreNLP [28] parser which is a suite of NLP software tools:

```
(ROOT
  (S
    (NP (DT The) (NN user))
    (VP (VBZ logs)
      (PP (IN into)
        (NP (DT the) (NN system))))))
```

When parsing a sentence, its parts are labeled at clause level such as "S" which means simple declarative clause and at phrase level such as "NP", "VP" and "PP" which mean noun phrase, verbal phrase and prepositional phrase, respectively. Based on the used tagger, leaf nodes of the parsed tree such as "logs" have a depth of zero, tagged words such as "VBZ logs" have a depth of 1 and phrasal nodes such as "VP (VBZ logs)" have a depth greater than or equal 2.

Then, for each separate parsed sentence from the scenario, the algorithm checks whether it includes a simple sentence structure of "Subject" part and "Predicate" part. The subject includes a noun or a pronoun in addition to the words describing it. The predicate part includes the verb in addition to other words telling more about the subject. Furthermore, a "simple subject" should be included in the subject part as noun and a "simple predicate" should be included in the predicate part as verb. So, the parsed sentence must be a simple declarative clause "S" and include a noun phrase "NP" followed by verbal phrase "VP" and the verbal phrase "VP" must include at least one noun phrase "NP" (lines 7-9). If the parsed sentence does not satisfy this rule, the algorithm marks it as invalid, informs the system user to change the sentence structure and continues processing the subsequent steps. The valid structure of the parsed sentence helps in identifying the main parts of the sentence, such as objects, operations and messages exchanged among objects.

Then the algorithm declares and initializes the "parts[]" buffer, which is an array of four elements of type text to store data about participating objects, operations performed and messages passed among objects. The data of "parts[]" provides important information for producing a specialized script that will be added to "umlSequenceScript" and "umlClassScript" buffers.

After that, a semantic analysis is performed in order to identify the main parts of the parsed sentence, such as the actor that initiates the scenario and the participating objects, their interactions and the named messages exchanged among them during their interaction. There are usually two participating objects; one is named "caller" that makes a request and the other is named "receiver" that receives the request from the former. Determining the interactions among objects is also defined in this step. The type of interaction that is always defined as an operation between objects is analyzed in addition to type of data passed among these objects.

To define the "caller" object, the labeled "NP" subtree of the main parsed tree "S" is searched for words the tags of which start with "NN" label, then their concatenation is stored in "parts[0]" buffer. The searched labels are "NN", "NNS", "NNP" and "NNPS" which represent singular, plural, singular proper and plural proper nouns, respectively. It is important to note that there may be more than one word that describes the "caller" object such as "main screen" (lines 11-12). The "NP" subtree is also searched for a word the tag of which starts with "NNP" label to consider it as an actor and if it is found, it will be stored in "actor" buffer (line 13). Note that the actor is set once for the overall scenario which always represents a role that initiates the scenario.

Then, the labeled "VP" subtree of the main parsed tree "S" is firstly processed to capture the operation between the "caller" and the "receiver" objects by searching a word the tag of which starts with "VB" label and storing it in "parts[2]" buffer (line 14). The labels are "VB", "VBD", "VBG", "VBN", "VBP" and "VBZ" which represent base form, past tense, gerund or present participle, past participle, non-3rd person singular present and 3rd person singular present verbs, respectively. After that, if the "VP" subtree of the main tree "S" includes only one "NP" subtree, search it for the "NN" tagged words, consider their concatenation as a "receiver" object and store it in the "parts[1]" buffer. However, if it includes more than one "NP" subtree, consider the "NN" tagged words of the first "NP" as the message passed between the interacting objects, store their combination in "parts[3]", consider the "NN" tagged words of the second "NP" as the "receiver" object and store their combination in "parts[1]".

3) After the loop of sentence processing finishes, the four stored elements in "parts[]" buffer, which are the "caller" object, the "receiver" object, the operation and the message, are used to construct a one-line specialized script that will be added to "umlSequenceScript" and "umlClassScript" buffers. This script will be fed later to the PlantUML [29] plugin for generating the required UML diagrams. This script has the format of "caller -> receiver : operations message". For example, the result of processing this scenario sentence "Registrar selects addition from options’ screen" is "Registrar -> OptionsScreen : Selects Addition". This script structure follows the format invented by the PlantUML tool.

4) At the end of processing the whole scenario, if an actor is found, it will be added as a prefix to the "umlSequenceScript" buffer content. As an example, for the following scenario:

> Student selects "register for a course" from main menu.
> Application retrieves courses from database.
> Software presents courses into courses list window.
> Student selects courses from the courses list window.
> Student submits the selected courses.
> Application saves selected courses into database.

After processing the scenario with our proposed algorithm, the complete script for the sequence diagram is as follows:

```
Actor Student
Student -> MainMenu : Selects RegisterCourse
Application -> Database : Retrieves Courses
Application -> CoursesListScreen : Shows Courses
Student -> CoursesListScreen : Selects Courses
Student -> SelectedCourses : Submits
Application -> Database : Saves SelectedCourses
```

And the complete script for the class diagram is as follows:

```
Student o-- MainMenu
MainMenu : Selects ()
Application o-- Database
Database : Retrieves ()
Application o-- CoursesListScreen
CoursesListScreen : Shows ()
Student o-- CoursesListScreen
CoursesListScreen : Selects ()
Student o-- SelectedCourses
SelectedCourses : Submits ()
Application o-- Database
Database : Saves ()
```

It is important to denote that the sequence diagram is considered as the basis for drawing the class diagram.

4. SYSTEM DESIGN AND IMPLEMENTATION

We have developed a system for implementing the previously stated algorithm that essentially processes scenario-based requirements written in English text, then automatically draws UML diagrams. The system is named AGUML and has a capability of reading scenario-based user requirements. It also has a capability of applying NLP on this text and making syntactical and semantical analysis to produce UML diagrams. The main components of the system architecture are shown in Figure 2. Descriptions of the GUML system components are in the following subsections.

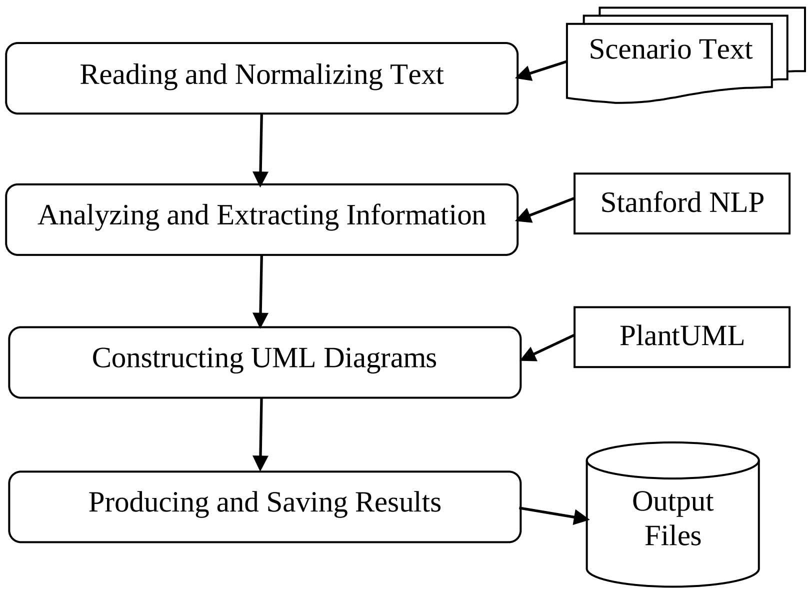

Reading and Normalizing Text
Scenario Text
Analyzing and Extracting Information
Stanford NLP
Constructing UML Diagrams
PlantUML
Producing and Saving Results
Output Files

Figure 2. AGUML system architecture.

4.1 Reading and Normalizing Scenario Text

This component takes a scenario from user as text input. The user can directly write the scenario text or retrieve it from an existing file. Then, the text is normalized to convert the equivalent words into a uniform format.

4.2 NLP of Normalized Text

It firstly separates each sentence of the scenario text, produces POS tagging and parses this sentence to produce a tree in order to help in syntactic and semantic recognition. This component also checks the validity of sentence structure by deciding whether it conforms to the form of noun phrase followed by a verbal phrase and the verbal phrase has at least one noun phrase. And hence, the user will be informed of invalid scenario sentences.

4.3 Analyzing and Extracting Information

This component takes the tagged and parsed text as input from the previous component, then defines the objects, their interactions and the type of data passed among them. It mainly analyzes the noun phrase and the verbal phrase in each parsed scenario text and the tagged words of each phrase. The actor of the scenario is defined in this module if it exists. If the NLP results produce more than one actor, only one equivalent uniform actor will be assigned, because the scenario is always initiated by one actor. Actor can be identified by searching proper noun in the first part (noun phrase) of the scenario sentence. The analysis and extraction of this important information is achieved with the help of Stanford CoreNLP software tools.

4.4 Constructing UML Diagrams

This component takes information about diagrams from the previous component and produces a specialized script. This script is next processed by the included PlantUML third-party component to draw sequence and class diagrams.

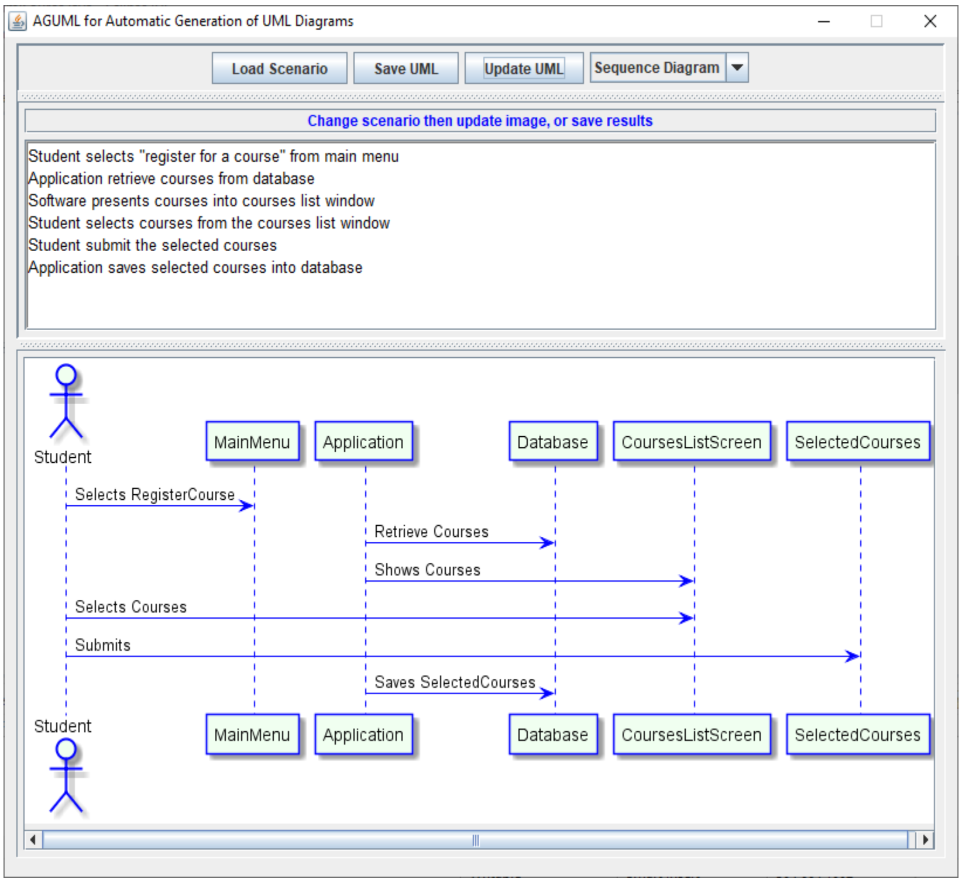
4.5 Producing and Saving Results

This module helps the user save the updated scenario text, sequence diagram and class diagram in order to be used for documenting purposes that will support the subsequent tasks of the software development process. It also permits for updating these results later.

We have implemented the system using Java programming language with the help of Eclipse IDE [30].

5. Experimental Results

This section provides the analysis of our experimental results. The experiments were mainly conducted to check the accuracy of our designed algorithm that has been implemented in the AGUML system. The main window of AGUML software tool is shown in Figure 3. Using this window, the user can write a scenario as separate sentences of English text or can load it from previously stored text file. When pressing the button "Update UML", our software tool will process the scenario text and produce the UML diagrams automatically. AGUML can show sequence diagram, class diagram or both of them simultaneously based on user selection. Figure 3 shows how sequence diagram appears after processing a scenario. The actor appears on the left side of the diagram and it is represented using stick person symbol. The "caller and "receiver" objects appear on the top and on the bottom of the diagram and they are represented as annotated rectangles. The name of the operation and the message passed between objects are represented using leftward arrow symbol that links the "caller" object and the "receiver" object.

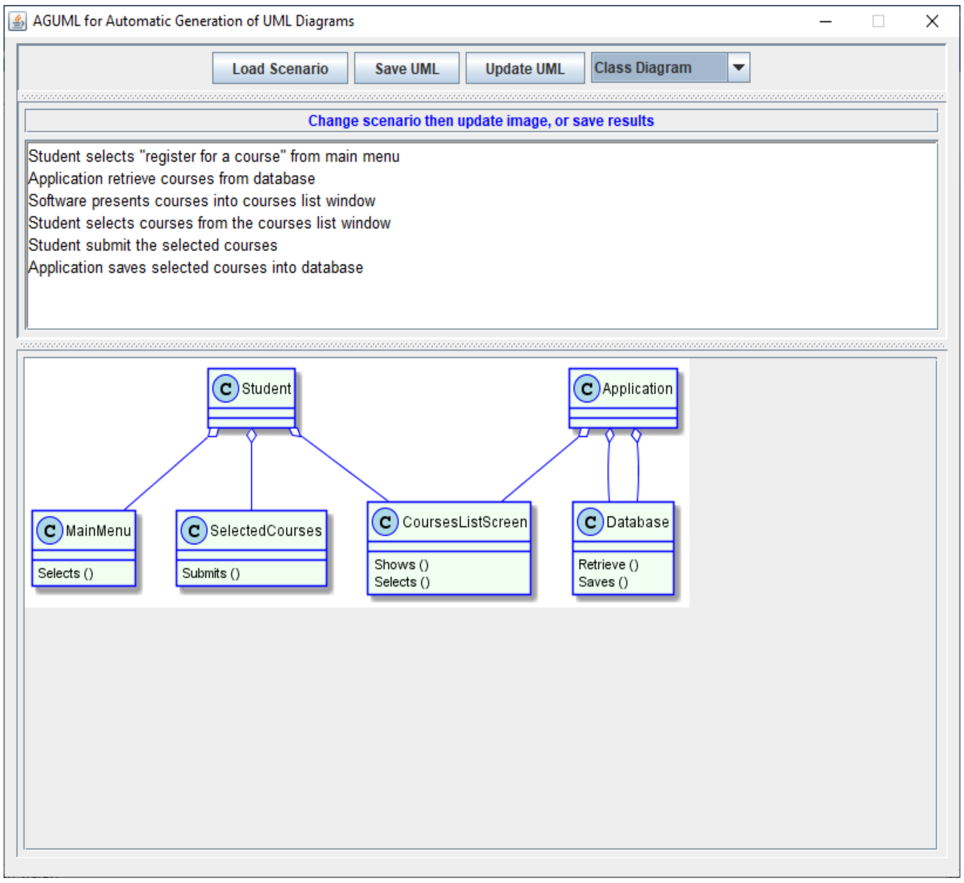


Student selects "register for a course" from main menu
Application retrieve courses from database
Software presents courses into courses list window
Student selects courses from the courses list window
Student submit the selected courses
Application saves selected courses into database

Figure 3. Generation of UML sequence diagram from scenario text.

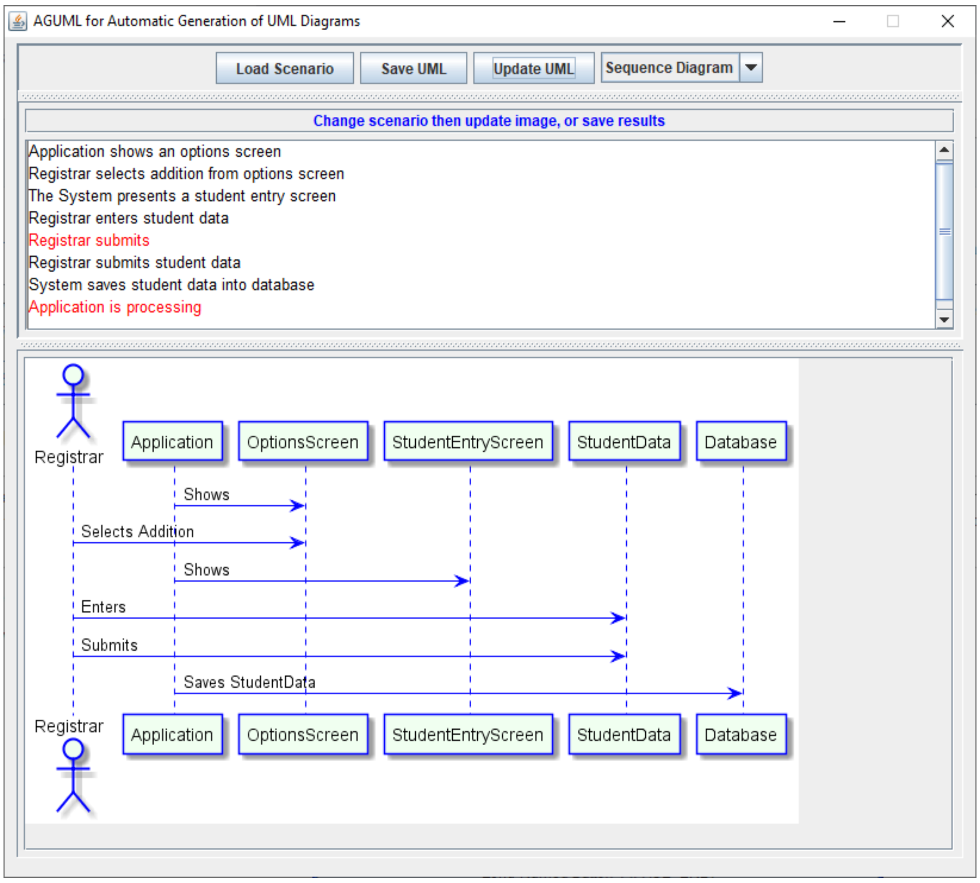
The class diagram of the same scenario is shown in Figure 4. As said previously, the components of the class diagram are constructed depending on the developed sequence diagram. So, the same classes identified in the sequence diagram will also appear in the new class diagram. Furthermore, the operations between objects shown in the sequence diagram will be added as operations inside the classes of the class diagram. They are actually added inside the class which provides the functionality for the requesting class; in other words, in the class where the head of leftward arrow ends in the sequence diagram.

AGUML also can provide the user with the ability of storing important results into files in order to be used in software documentation as in well as any future requirements updates. These results include updated scenario text, UML-generated diagrams and UML input script. Furthermore, when processing



Student selects "register for a course" from main menu
Application retrieve courses from database
Software presents courses into courses list window
Student selects courses from the courses list window
Student submit the selected courses
Application saves selected courses into database

Figure 4. Generation of UML class diagram from scenario text.



Application shows an options screen
Registrar selects addition from options screen
The System presents a student entry screen
Registrar enters student data
Registrar submits
Registrar submits student data
System saves student data into database
Application is processing

Figure 5. Invalid scenario sentences are shown in red text.

scenario text, AGUML can notify user with any invalid English sentence in order to help user in correcting the invalid sentence, as shown in Figure 5. However, AGUML has the ability of generating UML diagrams for only valid scenario sentences.

To assess the accuracy of AGUML and the designed algorithm, we have evaluated it against five different real-life scenarios collected from different sources in the literature. The first scenario describes "add student" use case and the second scenario describes "register for a course" use case and they are part of defined requirements for educational and registration services in a university [4]. The third scenario describes "check out item" use case and the fourth scenario describes "view file content" use case and they are part of defined requirements for videos' store services [5]. The fifth scenario defines the steps for verifying user by login name and password [31]. Each scenario includes a hybrid of simple, average and complex English sentences and they are labeled from "SC1" to "SC5" in Table 1.

After checking the sentence validity for each scenario, we have considered five types of criteria for evaluating AGUML which are actor identification, "caller" object identification, "receiver" object identification, operation identification and message identification.

Firstly, we have developed sequence diagrams manually for each scenario based on our experience and provided counts for each one of the five elements in each individual diagram. These counts are found inside the columns labeled with "M" in Table 1. Similarly, we have provided counts for each of the five elements in each individual scenario generated automatically by AGUML, where their identifications match correctly the manual identifications. The counts of automatic identifications are found inside the columns labeled with "A" in Table 1. The accuracy is then calculated by dividing the count of automatic generation by the count of manual generation for each element [14].

Table 1. Accuracy results of AGUML for automatic generation of UML diagrams.

Scenario	**SC1**		**SC2**		**SC3**		**SC4**		**SC5**		**Accuracy**
Element	M	A	M	A	M	A	M	A	M	A	
Actor	1	1	1	1	1	1	1	1	1	0	80%
Caller object	3	3	2	2	2	2	2	1	2	2	90%
Receiver object	5	5	4	4	5	3	7	7	6	5	88%
Operation	7	7	6	6	6	4	7	6	7	7	90%
Messages	2	2	5	5	0	0	1	0	1	0	77%

As shown in Table 1, the accuracy ranges from 77% to 90%, where the "caller" object identification and operation identification have the highest accuracy scores.

After reviewing the POS tree for some sentences, it is noticed that the adopted tagger tool produces tags for some words in a manner that is different from our intension. For example, it considers "user" in "The user enters a username and password" sentence of "SC5" and "SC4" as "NN" and hence, it is not considered as an actor. Also, the sentence "Clerk swipes bar code" of "SC3" was tagged as:

```
(ROOT
  (S
    (VP (VB Clerk)
      (NP (JJ swipes) (NN bar) (NN code)))))
```

This means that the tagger considers "Clerk" word as verb and "swipes" word as adjective and hence, the AGUML reports it as an invalid sentence. Moreover, "Application stores record" sentence of the same scenario was tagged as:

```
(ROOT
  (NP
    (NP (NN application) (NNS stores))
    (NP (NN record))))
  (NP (NP) (NP)) stores as NNS
```

And hence, the tagger does not consider it as an adequate sentence structure labeled with "S". Although the AGUML accuracy is affected by the tagger behavior in some cases as seen in these examples, it is generally accepted, especially in the case of building UML sequence diagrams.

6. Conclusion and Future Work

In this paper, we have proposed and developed a system called AGUML for automatic generation of UML diagrams from scenario-based user requirements written in English natural language. We have designed an algorithm and implemented it in the AGUML. It has a capability of producing syntactical, lexical and semantic analysis for extracting important information for identifying actors, classes/objects and their interactions as well as the messages exchanged among them. The main purpose of AGUML is to facilitate work performed in the analysis and design activities for the software being developed. AGUML has an interactive user interface to help user write a scenario directly or upload it from an existing file. Then, AGUML can process the scenario and produce the UML diagrams automatically. The resulting diagrams can be stored easily in files in order to be used later for the purposes of software documentation. The accuracy of AGUML and the designed algorithm has been tested using five different scenarios.

An extension to this work is needed for enhancing the designed algorithm in order to permit the user for writing less restrictive scenario text and considering other variants of scenario main flow such as alternative flow. The capability of our proposed system can be also enhanced in order to produce other UML diagrams.

References

[1] D. K. Deeptimahanti and M. A. Babar, "An Automated Tool for Generating UML Models from Natural Language Requirements," Proc. of the IEEE/ACM Int. Conf. on Automated Soft. Eng., pp. 680-682, Auckland, New Zealand, 2009.

[2] I. F. Alexander and N. Maiden, Scenarios, Stories, Use Cases: Through the Systems' Development Life-cycle, ISBN: 978-0-470-86194-3, John Wiley & Sons, 2005.

[3] H. Gomaa, Software Modeling and Design: UML, Use Cases, Patterns and Software Architectures, DOI: 10.1017/CBO9780511779183, Cambridge University Press, 2011.

[4] T. Quatrani, Visual Modeling with Rational Rose 2002 and UML, 3rd Edition, Pearson Education India, ISBN-10: 0201729326, 2002.

[5] E. J. Braude, Software Design: From Programming to Architecture, John Wiley and Sons, 2004.

[6] R. Elmansouri, S. Meghzili, A. Chaoui, A. Belghiat and O. Hedjazi, "Integrating UML 2.0 Activity Diagrams and Pi-Calculus for Modeling and Verification of Software Systems Using TGG", Jordanian Journal of Computers and Information Technology (JJCIT), vol. 06, no. 04, pp. 326-344, Dec. 2020.

[7] R. J. Abbott, "Program Design by Informal English Descriptions," Communications of the ACM, vol. 26, no. 11, pp. 882-894, 1983.

[8] E. Buchholz, A. Düsterhöft and B. Thalheim, "Capturing Information on Behavior with the RADD-NLI: A Linguistic and Knowledge-based Approach," Data & Knowledge Engineering, vol. 23, no. 1, pp. 33-46, 1997.

[9] S. Nanduri and S. Rugaber, "Requirements' Validation *via* Automated Natural Language Parsing," Journal of Management Information Systems, vol. 12, no. 3, pp. 9-19, 1995.

[10] A. Oliveira, N. Seco and P. Gomes, "A CBR Approach to Text to Class Diagram Translation," Proc. of the TCBR Workshop at the 8th European Conference on Case-based Reasoning, Turkey, Sep. 2006.

[11] H. M. Harmain and R. Gaizauskas, "CM-builder: A Natural Language-based Case Tool for Object-oriented Analysis," Automated Software Engineering, vol. 10, no. 2, pp. 157-181, 2003.

[12] S. P. Overmyer, L. Benoit and R. Owen, "Conceptual Modeling through Linguistic Analysis Using LIDA," Proceedings of the 23rd IEEE Int. Conf. on Soft. Eng. (ICSE 2001), pp. 401-410, May 2001.

[13] M. Clavel, M. S. E. González and V. T. da Silva, "The MOVA Tool: A Rewriting-based UML Modeling, Measuring and Validation Tool," Proc. of the XII JISBD, pp. 393-394, Spain, 2007.

[14] I. S. Bajwa, A. Samad and S. Mumtaz, "Object Oriented Software Modeling Using NLP Based Knowledge Extraction," European Journal of Scientific Research, vol. 35, no. 1, pp. 22-33, 2009.

[15] M. Ibrahim and R. Ahmad, "Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques," Proc. of the 2nd IEEE Int. Conf. on Computer Research and Development, pp. 200-204, Kuala Lumpur, Malaysia, May 2010.

[16] H. Herchi and W. B. Abdessalem, "From User Requirements to UML Class Diagram," Proc. of the

International Conference on Computer Related Knowledge, arXiv Preprint arXiv:1211.0713, 2012.

[17] P. More and R. Phalnikar, "Generating UML Diagrams from Natural Language Specifications," International Journal of Applied Information Systems, vol. 1, no. 8, pp. 19-23, 2012.

[18] H. G. Perez-Gonzalez and J. K. Kalita, "Automatically Generating Object Models from Natural Language Analysis," Proc. of the Companion of the 17th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Lang. and Appl. (OOPSLA '02), pp. 86-87, Seattle, USA, Nov. 2002.

[19] M. Elallaoui, K. Nafil and R. Touahni, "Automatic Transformation of User Stories into UML Use Case Diagrams Using NLP Techniques," Procedia Computer Science, vol. 130, pp. 42-49, 2018.

[20] S. Jabbarin and N. Arman, "Constructing Use Case Models from Arabic User Requirements in a Semi-automated Approach," Proc. of the IEEE World Congress on Computer Applications and Information Systems (WCCAIS), pp. 1-4, Hammamet, Tunisia, Jan. 2014.

[21] M. Elallaoui, K. Nafil and R. Touahni, "Automatic Generation of UML Sequence Diagrams from User Stories in Scrum Process," Proc. of the 10th IEEE International Conference on Intelligent Systems: Theories and Applications (SITA), pp. 1-6, Rabat, Morocco, Oct. 2015.

[22] N. Alami, N. Arman and F. Khamayseh, "Generating Sequence Diagrams from Arabic User Requirements Using MADA+ TOKAN Tool," The International Arab Journal of Information Technology, vol. 17, no. 1, pp. 65-72, Jan. 2020.

[23] I. Alexander and N. Maiden, "Scenarios, Stories and Use Cases: The Modern Basis for System Development," Computing and Control Engineering, vol. 15, no. 5, pp. 24-29, 2004.

[24] A. G. Sutcliffe, N. A Maiden, S. Minocha and D. Manuel, "Supporting Scenario-based Requirements' Engineering," IEEE Transactions on Software Engineering, vol. 24, no. 12, pp. 1072-1088, 1998.

[25] Spark Systems, "Writing Use Case Scenarios for Model Driven Development," [Online], Available: https://sparxsystems.com/downloads/quick/writing-structured-use-case-scenarios-mdd.pdf, 2010.

[26] I. Sommerville, Software Engineering, 10th Edition, Addison-Wesley, 2015.

[27] I. Jacobson, Object-oriented Software Engineering: A Use Case Driven Approach, Pearson Education India, 1993.

[28] Stanford CoreNLP, "A Suite of Core NLP Tools," [Online], Available: http://stanfordnlp.github.io/CoreNLP.

[29] PlantUML, [Online], Available: http://plantuml.com.

[30] Eclipse Foundation, [Online], Available: https://www.eclipse.org.

[31] G. Schneider and J. P. Winters, Applying Use Cases: A Practical Guide, Pearson Education, 2001.

ملخص البحث:

تعدّ أدوات النّمذجة الفعالة للبرمجيات ضرورية جداً للتّحقيق النّاجح لأنشطة هندسة البرمجيات، وبخاصة في مرحلة التّحليل والتّصميم. وإنّ أتمتة هذه الأدوات تسهل العمل، وتجعله أعلى إنتاجيةً، وتخفّض زمن التّطوير وكلفته. وتهدف هذه الورقة الى تطوير طريقةٍ والتحقق منها -بالإضافة الى أداةٍ برمجية- للتّوليد الاتوماتيكي لمخططات لغة النّمذجة الموحدة عند اتباع النّهج الموجَّه نحو الهدف في التطوير. ويتم توليد هذه المخططات من المتطلبات المبنيّة على السيناريوهات؛ من أجل تسهيل عملية النّمذجة. وهكذا يتم تحديد نموذجٍ من المتطلبات المبنية على السيناريوهات ومكوناته وبناؤها. ثم يتم تصميم طريقةٍ تتضمن خوارزمية وتطبيقها بناءً على معالجة اللغات الطبيعية لتوليد مخططات لغة النّمذجة الموحدة أوتوماتيكياً من المتطلبات المبنية على السيناريوهات، وتشمل مخططات التتابع ومخططات الصِّنف. وجرى التحقق من قدرة الطريقة المقترحة وأدائها وفوائدها، وكذلك الأداة البرمجية، عن طريق نتائج تجارب عملية أجريت لهذا الغرض.