Ayad Tareq Imam* and Ayman Jameel Alnsour

The Use of Natural Language Processing Approach for Converting Pseudo Code to C# Code

https://doi.org/10.1515/jisys-2018-0291
Received July 3, 2018; previously published online April 16, 2019.

Abstract: Although current computer-aided software engineering tools support developers in composing a program, there is no doubt that more flexible supportive tools are needed to address the increases in the complexity of programs. This need can be met by automating the intellectual activities that are carried out by humans when composing a program. This paper aims to automate the composition of a programming language code from pseudocode, which is viewed here as a translation process for a natural language text, as pseudocode is a formatted text in natural English language. Based on this view, a new automatic code generator is developed that can convert pseudocode to C# programming language code. This new automatic code generator (ACG), which is called CodeComposer, uses natural language processing (NLP) techniques such as verb classification, thematic roles, and semantic role labeling (SRL) to analyze the pseudocode. The resulting analysis of linguistic information from these techniques is used by a semantic rule-based mapping machine to perform the composition process. CodeComposer can be viewed as an intelligent computer-aided software engineering (I_CASE) tool. An evaluation of the accuracy of CodeComposer using a binomial technique shows that it has a precision of 88%, a recall of 91%, and an F-measure of 89%.

Keywords: ACG, I-CASE, NLP, SRL, thematic role, verb classification.

1 Introduction

Software manufacturing can be enhanced in terms of both its quality and quantity by means of computer-aided software engineering (CASE) tools, which are a set of software systems for fully or partially automating certain activities of the software development process. These tools are available either separately or as a package. A well-known example of CASE tools is Rational Rose. Figure 1 illustrates a typical architecture for a set of CASE software tools [39, 51].

The composition of a program is a fundamental phase in the software development life cycle (SDLC) that can be automated via code generation CASE tools [39]. Software engineering (SE) terms this phase implementation, and it follows the design phase. The conversion of a certain design to a programming language code is a relatively straightforward task compared to the other software development tasks [51].

Composing the source code of a program (or composing any program) is one task among many associated with computer programming activity, such as testing, debugging, and maintaining a source code. In addition to being an engineering discipline, good program writing is also viewed as an art [11]. In general, there are two techniques for writing the source code of a program: updating an existing code and creating a new source code. Obviously, the composition of source codes requires proficiency in specialized algorithms that are based on a knowledge of the application area [21].

***Corresponding author: Ayad Tareq Imam,** Department of Computer Science, Faculty of Information Technology, Isra University, Amman 11622, Jordan, e-mail: alzobaydi_ayad@iu.edu.jo. https://orcid.org/0000-0002-9942-4772
Ayman Jameel Alnsour: Department of Computer Engineering, College of Computer Engineering and Sciences, Prince Sattam Bin Abdulaziz University, Alkharj, Kingdom of Saudi Arabia

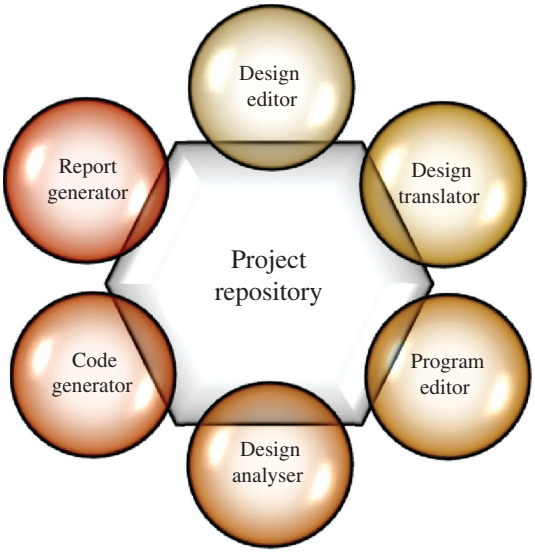


Figure 1: Architecture of CASE Software Tools.

The composition of a program is realized by converting an algorithm (or, more generally, a design) into a programming language code. The choice of an algorithm to solve a given problem is the responsibility of professional programmers [11].

Conversion of an algorithm, which is initially created as pseudocode, flowchart, or another form, into a programming language code is achieved manually or automatically [3]. The problem addressed in this paper is how to automate the process of composing a program by converting the pseudocode to the C# programming language. The solution proposed here is the use of the semantic role labeling (SRL) of natural language processing (NLP) in an automatic code generator (ACG).

1.1 Pseudocode

This is an informal (i.e. in natural language rather than in a programming language), formally styled, and detailed description of an algorithm. Pseudocode forms of a solution are not executable on computers, although they form a kind of template for developing an executable program by converting it to a certain programming language [53].

As it takes the form of natural language, the pseudocode offers the software development team a tool to verify that the solution matches the specifications of the design, with no need to learn a specific description language. The discovery of logical errors at this stage costs less than discovering such errors in the subsequent stages of the development process, and the pseudocode is therefore considered to be a CASE (non-software) tool [48].

On the other hand, unlike programming languages and other artificial languages such as Math, the pseudocode has no defined set of words, and it is left to the developer to choose words that can deliver a particular solution. As there is no agreed, standardized style or format, forms of pseudocode vary widely from each other. The representation of input, output, and processing activities can be achieved using any word that serves this goal. For example, the word “input” can be used instead of the word “read”, or the word “display” can be used in place of the word “write” [37].

Although it has a structure, the pseudocode tends to be in the style of natural language and, hence, inherits problems related to the use of natural language in everyday communication, such as the ambiguity resulting from different interpretations of a word, a statement, or speech, in general. The problems of the absence of standards and the very few rules used by the pseudocode are two of its main shortcomings [48]. As the pseudocode is not limited to particular words, the mapping of the syntax to C# semantics (which are C# statements) poses a challenge. Maintaining a lookup table would not be helpful, due to the fact that

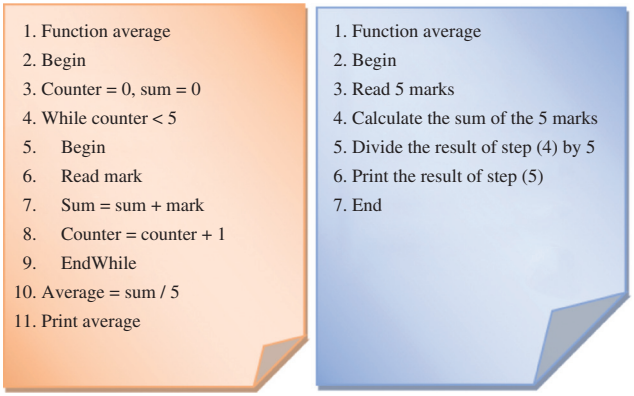


Figure 2: Two Different Pseudocodes for a Given Algorithm.

words other than those defined in the lookup table may be used, and this is critical problem that needs to be addressed. Hence, NLP should be used. Figure 2 illustrates two different pseudocodes for a single algorithm.

1.2 C# Programming Language

C# is a high-level programming language that was designed and developed by Microsoft. Both the International Standards Organisation (ISO) and the European Computer Manufacturers Association (ECMA) have approved this language. C# is analogous to Java in that it is a general-purpose, object- and component-oriented part of the .Net Framework, which is a platform that supports the writing of various types of modern applications, such as windows, web applications, and web services and is designed for common language infrastructure (CLI) [7].

C# is distinguished from closely related, traditional, high-level languages, such as C, C++, and Java, by the extra features of the constructs it has, such as automatic garbage collection, a standard library, properties and events, delegation and event management, indexers, simple multithreading, integration with windows, and others [7].

Of the various specifications of C#, this research is primarily interested in its syntax and constraints. This is because an algorithm represented by pseudocode aims to describe a plan for a process, which corresponds to a method in C# terminology. The syntax and constraints of C# can be found in Ref. [7].

1.3 Automatic Code Generator

ACGs are a class of CASE software tools that automate the process of composing a program or, in other words, the use of programs to generate source code that humans would otherwise have to write [52]. The use of ACG saves time and effort in addition to improving the software quality, productivity, consistency, accuracy, and abstract coding. One of the most popular examples of automatic code generation is the conversion of a designed graphical user interface (GUI) to an executable code in a visual programming environment such as Visual Studio [39].

ACGs are either passive or active. A passive code generator produces a code that needs some sort of human adjustment or modification, while an active code generator is embedded in the software development process, and its execution is repeated in order to generate a new code [23, 29].

Forward/reverse engineering software tools (which are integrated with software modeling tools), code wizards, and compilers are typical examples of the approaches used by an ACG to accomplish its function, regardless of its type. Examples of software that convert an algorithm into a programming language code include the conversion of an algorithm to code by the method in Ref. [3], Code Master (algorithm-to-code converter) presented in Ref. [43], the AthTek Flowchart to Code [50], Flowgorithm [19], and many more.

Although the available ACGs have had remarkable successes, most ACG software tools require human intervention to design a solution (algorithm or a system) as a prior step. This shortcoming is accepted as

normal, as design is a creativity-based skill that is an exceptionally hard task to automate. Based on this fact, some have expressed doubts about the possibility of developing software tools to automate the design task [15].

1.4 Natural Language Processing

NLP is a type of artificial intelligence (AI) processing that aims to allow a computer to understand human natural language. NLP extends text processing beyond simple syntactic processing to major and critical semantic processing, which is a human natural ability [27, 32]. There are several different AI approaches that are used by NLP applications to accomplish the task of understanding (inferring) the intended meaning of human speech. A traditional rule-based approach involves inference rules that use predefined criteria (conditions) to map the syntax to a suitable semantics [32], while a connectionist approach uses a learning strategy to develop a mapping (or classifying) machine. With the advancements in machine learning applications, more flexible, intuitive learning algorithms were defined that can allow a computer to discover the intent of a speaker. Deep learning, as it is termed, requires enormous amounts of labeled data (called examples) to train the computer prior to utilizing it for a particular application. This training process aims to allow the computer to automatically discover relevant correlations among input patterns and output classes. This strategy is similar to the way a baby begins to learn a human language. NLP uses deep learning to apply understanding to the developed program [32].

It is worth mentioning here that the development (or composition) of an algorithm is another task that is classified as problem solving. Although the composition of an algorithm and a program are separate tasks in a complex programming project, they are combined in simpler ones [11]. This combination is considered as one of algorithm's notations (representation techniques) [48], and hence, this composing tends to be a problem-solving issue (i.e. design of a solution) more than a compiling issue (i.e. converting a design into a source code). Software for automation of a design activity is classified as an intelligent computer-aided software engineering (I-CASE) tool [24].

This paper contributes ACGs using the SRL aspect of NLP together with a semantic rule-based (logic-based) approach to generate a source code (or compose a program). The results of our proposed ACG are more detailed than those resulting from ACGs using visual programming integrated development environment (IDE) SW tools that convert the design of a GUI to a code, which are limited to the header of the method and empty braces in the method's body, and do not involve programming statements in the body of the method.

2 Related Approaches and Works

An investigation of previous related works shows that it is possible to develop programs for converting (translating) a pseudocode algorithm into a specific programming language [3, 53]. An interesting list of code generation tools is given in Ref. [52] as a part of comparison study between code generation tools. In this comparison, especially the technical one, the data model is used as an input technique in these code generation tools. Unfortunately, their processing techniques are not mentioned.

A *text processing approach* (Text processing is the electronic creation or manipulation of a text that makes no use of NLP phases [27].) is a well-known and possibly the simplest approach for accomplishing the task of code generation in terms of a translation from one linguistic (possibly artificial) form to another. In this approach, a lookup table is maintained, and a blind mapping is performed based on this table. In practice, this approach is not feasible, as the translation of a text from one linguistic form to another requires more than a blind mapping [27].

Model-driven engineering (MDE) (MDE is a synonym for model-driven architecture (MDA), which uses models as a major artifact for software development, unlike processes that utilize source code as a major artifact [30].) methods and tools are used to automate the generation of a software code [30]. One example of a work based on this approach uses a class diagram in unified modeling language (UML) to generate source

codes for programs [49]. Although it has been used, this approach requires developers to have significant abstraction capacity to develop MDE-based tools, so that many users can take advantage of these tools [16].

Generating a programming code from a pseudocode (as a source code) can be viewed as a code generation task that can be achieved by a machine translation (MT) system. This approach is distinguished by the use of an interlingual (intermediate) representation for the source code, which is used to generate the target code. This approach is a framework that encompasses two monolingual components: the analysis process, which works on the source language to produce an intermediate form (interlingual form), and the generation process, which works on interlingual forms and produces the target language form [54]. A compiler software is an example of an interlingual MT system. Although it aims to produce an executable code, i.e. a low-level language (LLL) from the high-level language (HLL) of a program [1], a compilation approach (i.e. one that is used to develop a compiler software) can also be used to generate one HLL program from another, as shown in Refs. [4] and [22]. As we noted, while the good structuring process it has, this approach works successfully with artificial languages (that are the programming languages), but it is hard to be used successfully with natural language like the pseudocode. In our case, as the source language, i.e. the pseudocode, is in fact a natural language form, it is important to focus on an AI-based (or knowledge-based) interlingual MT, which includes NLP in the form of lexical, syntax, semantic, and pragmatic knowledge, in addition to knowledge acquisition and an optimization process.

Language-based MT (LBMT) is a type of MT that may be either lexeme-oriented or grammar-oriented. In a lexeme-oriented approach, translation is achieved by relying on the principle of lexical equivalence, in which the units of translation are words and specific phrases only [54]. In a grammar-oriented approach, the unit of translation is a set of structural attributes of the source text and is limited to intra-sentential structures. Despite the differences between these two types of LBMT approach, both give little insight into the use of context, representing a domain of discourse such as social, medical, or financial discourse [17, 27, 54].

In view of the huge body of hand-coded lexical knowledge, the semantic roles that are linked with certain syntactic patterns, and background knowledge (ontology and domain models), knowledge-based MT (KBMT) was defined to perform the translation process taking into account the meaning of the source text [14]. Figure 3 illustrates the standard knowledge-based MT system [54].

One example of a system that uses this approach is KANT. However, the potential achievements of KBMT depend on the presence of a well-defined model of conceptual domains across a diversity of cultures and a linguistic mapping model that can discover the meaning of syntactic forms [31].

In order to address the complexity of meeting the conditions for the KBMT's potential achievements, the example-based MT (EBMT) approach was proposed. The EMBT achieves a translation using collected examples of previous translation processes, for which these examples form a specific body of knowledge.

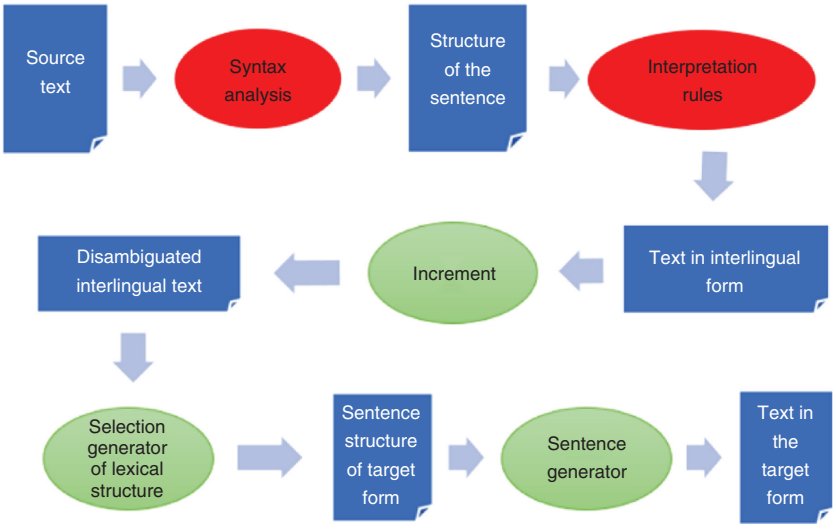


Figure 3: Knowledge-Based MT System.

Translation is performed using the similarity in the annotation of the surface descriptions that is assigned to each example that forms the knowledge. The surface description annotation encompasses, for example, patterns of word classes, certain strings, the dependencies of words [46], and frames of predicate [26] that are used for joining the translated units. EBMT gives an extended role to AI in composing a program, as can be seen in the artificial neural network (ANN) (ANN is a computational approach based on mapping an input pattern to one of a set of defined decisions [32].) approach. Although it is a classification approach used particularly with distorted data, ANN was used to compose a source code from a pseudocode, as described in Refs. [37] and [34]. As the EBMT's mapping process is implemented using the principle of approximate reasoning, it means that instead of a single exact translation, degrees of acceptability (based on probability and distance measures) are used to select a translation among a set of possible translations [20]. Of course, more alternatives are produced as a more complex solution is required, which shows the complexities that EBMT should have to reach its goal.

In all types of MT listed above, the resources required to develop an interlingual MT system are, in general [54]:

- Linguistic lexicons, which are used in the lexical detection of tokens of the source text and in the lexical generation of the target text;
- Syntactic grammars, which are used to analyze the structures of the source text and to generate the structures of the target text;
- A conceptual lexicon related to a specific domain, which is used to understand the text by recognizing entities and events; and
- Semantic (projection) rules to define relationships between events and entities.

Logic-based approach, which is implemented as rule-based, experience-based, and case-based systems is another technique that have been used to develop converting application from a pseudocode to a programming code. Examples of these applications include a logic-based approach to reverse engineering tool production [13], the reuse assessor and improver system (RAIS) [42], a fuzzy logic-based approach for software testing [56], a tile logic-based approach for software architecture description analysis [2], an expert code generator using rule base and frame knowledge representation techniques [23], and many more. These works illustrate the diversity of the logic used (binary, fuzzy, etc) and how this diversity is utilized in developing logic-based systems. We noted that the problem with these works is that they do not consider NLP (especially the semantic part) as a main requirement in the converting process.

Deductive and inductive (Deductive reasoning is an approach that is used to prove the soundness of a theory. Inductive reasoning is an approach that starts with initial data and proceeds until a goal is reached [32].) approaches are two other forms of AI reasoning that are used to solve the problem of composing the code of a program in terms of the design of a solution [12, 15]. Based on these reasoning approaches, fully automated deductive programming (DP) and inductive programming (IP) can be used to generate parts of algorithms using UML diagrams and program synthesis. These parts of the algorithms are later used by an ACG in the process of composing a program code that includes loops or recursion. A semi-automatic induction approach, implemented as an intelligent agent, utilizes exemplary performance and end-user programming to identify recursive policies [15, 29]. The research work in Ref. [5] is a good example of the use of a machine learning approach to accomplish a repair task. Another AI approach, the *genetic algorithm (GA)*, is used also in composing programs. The AI programming system proposed in Ref. [10] is an interesting example that demonstrates the use of an AI approach to accomplish intellectual tasks such as composing a program. It is clear from the above that these approaches are used as parts of a whole process to automate the generation of a solution for a problem, and they do not convert pseudocode to programming code by their alone.

The NLP approach is an advanced topic of text processing that takes into account semantic processing to provide more flexibility when performing the mapping of text from one form to another. It is used with the KBMT and LBMT. This approach was used (partially) to develop an interpreter for converting algorithms written in natural English language to C code [35]. As we noted, the previous works based on this approach did not consider verb classification, thematic role, and SRL – the advanced topics of natural language semantic processing.

In this paper, the composition of a program is achieved using a hybrid approach that consists of SRL with logic-based (semantic) mapping rules. No previous related work has employed this approach, and this is, therefore, an original contribution of this paper. This approach is computerized to form the processing machine of a proposed ACG called CodeComposer, which is a system for composing programs from a pseudocode.

3 CodeComposer

In this paper, the proposed CodeComposer system is an MT system that translates pseudocode statements into programming language statements. The CodeComposer ACG is different from similar works in that it uses SRL and NLP with a semantic rule-based approach to generate a programming language source code. As illustrated in Figure 4, the processing flow of CodeComposer consists of several components, which are described below.

3.1 Natural Language Processing Step

In addition to conditional and repetition control statements, the pseudocode involves a set of natural language descriptions of the instructions that are used to write an algorithm [48]. While control statements (conditional and looping statements) are relatively clear and are similar to those used in a programming language, the units that should be focused on in the conversion process are the instructions, which are verbs in a linguistic sense.

A verb, which is a key part of the structure of a sentence, is used to identify a state or an event in which participants are involved, and hence, the meaning of a verb is considered key to the meaning of the sentence. As verbs are polysemous, the problem of resolving the lexical ambiguity of verbs can be tackled by considering their semantics [47]. In general, recognizing the semantics of a verb is a very difficult task due to the nature of verbs, which involves linguistic ambiguity (for example, a verb may have different semantics but a similar role in different phrases). A role-centered approach to lexical semantic representation has been suggested for studying the meanings of a verb. In this approach, the meaning of a verb can be represented using semantic role labels that are given to the verb's arguments. A well-known example of this approach, which reveals the difference between the verbs "break" and "hit", is given by Fillmore: "break" has the arguments (agent, instrument, object), while "hit" has the arguments (agent, instrument, place) [18]. Using a role-centered approach, several verb representation techniques were proposed such as grouping of verbs,

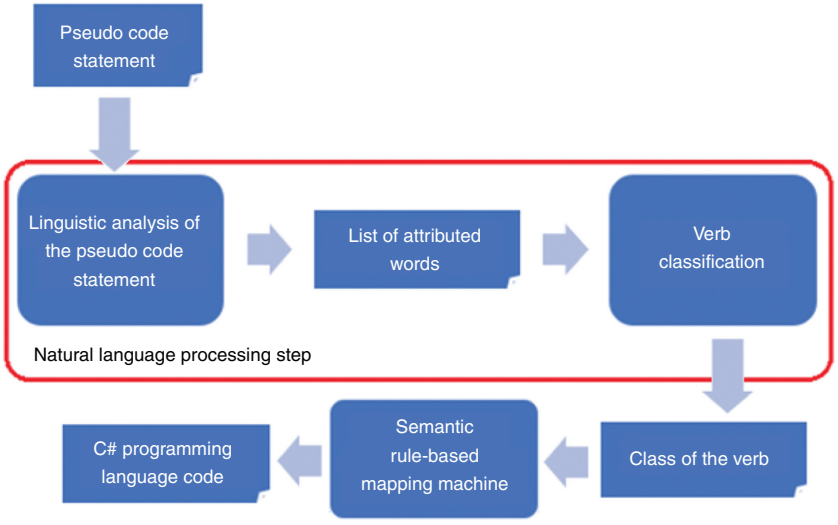


Figure 4: The Processing Flow of CodeComposer.

the argument roles of verbs, the structuring of instances' argument, a semantic frame, and semantic relationships. A corpus is available for the automatic recognition of the semantic roles of a verb, in which each item uses a one typical verb representation technique, for example:

- *The VerbNet (VN) corpus* uses a verb grouping representation technique. The VN corpus is the largest known online corpus and is constructed using Levin classes of verbs, which are enriched with new classified verbs [47].
- *The Proposition Bank (PropBank) corpus* uses a representation technique based on the argument roles of verbs. PropBank annotates 1 million English words with labels for the argument roles of verbs, defined by a supported lexicon [28].
- *The Noun Bank (NomBank) corpus* uses the structuring of instances' argument representation technique. NomBank supplies structures of instances argument for about 5000 English language nouns [33].
- *The FrameNet corpus* uses a semantic frame representation technique [8].
- *The WordNet corpus* uses a semantic relationships representation technique [55].

In our approach, as illustrated in Figure 4, a pseudocode statement forms the input to the natural language-processing stage of the pseudocode statement, which encompasses the following sub-steps:

1. Applying linguistic (lexical, syntax, and semantic) analysis to attribute each word of the pseudocode statement. This attribution of a pseudocode statement to words is important in order to isolate a verb and its parameters.
2. Verb classification, that is, searching for a *verb* entry that contains the syntax, semantics, thematic roles, and relations obtained in step (1), in which the *verb* entry class is the class of the verb within a pseudocode statement. The importance of this step lies in unifying the class of multi-form verbs to a single class, which can facilitate the mapping process to a C# statement.

The output of the natural language-processing step is a list (an internal representation) of a verb class and its attributes, which will be used as input for the following step, the semantic rule-based mapping machine.

3.1.1 Linguistic Analysis of the Pseudocode Statement

This step is achieved automatically using SRL software tools. Essentially, SRL is a natural language, high-level semantic process that is commonly used in information extraction, question answering, and similar systems. SRL is a process that discovers the predicate–argument structure of each predicate in a sentence. SRL identifies all components of a verb in a statement, where these components fill the semantic roles/thematic roles/theta roles that are required by the verb [27].

In this paper, an online software application called the SRL Demo [41] is used to reveal the thematic roles of a verb's components in a pseudocode of a solution. SRL typically uses symbolic notations rather than names to describe the semantic roles of the arguments; for example, [A0] represents an agent, and [A1] represents a patient. Table 1 illustrates the symbolic annotation of the semantic roles using syntactic forms. It should also be noted that it is difficult to determine a global set of semantic roles; hence, the number of semantic roles defined by linguistics may be different, and there may be between eight and 16 standard roles [40].

Table 1: Syntax, Semantic, and Symbolic Annotations.

Syntactic token	Semantic role	Annotation
Subject of a verb	Agent	[A0]
Object of verb	Theme (or patient)	[A1]
	Indirect object	[A2]
	Location of the event	AM-LOC
Verb	Action or event	[V]

To illustrate how this step is performed, consider the following pseudocode:

1. Function RecArea
2. Begin
3. Read length and width
4. Area = length * width
5. Print Area
6. End

Each pseudocode statement was manually (This was manually input using a technique involving copying from a pseudocode file and pasting it into a data field in the online SRL software application.) input into the SRL Demo software to perform a linguistic analysis of the statement. Figure 5 illustrates an example that represents the results of analyzing the pseudocode statements: "read length and width".

Note the similarity in the analysis of the (lexical and semantic) information in both statements, despite the different verbs used. Note also that the verb has no semantic annotation, which requires the identification of the verb's class to be sought. These annotated words (tokens) are used to identify the class of verb.

The results of the linguistic analysis are structured as a matrix, in which each row contains analysis information about a word in the submitted pseudocode statement. Figure 6 illustrates the structure of the matrix, which has a cell for each word along with its lexical annotation, semantic annotations, and a cell for the class of the verb, which is filled in later in the identification process. The size (number of rows) of the matrix will differ according to the size (number of words) of the analyzed pseudocode statement.

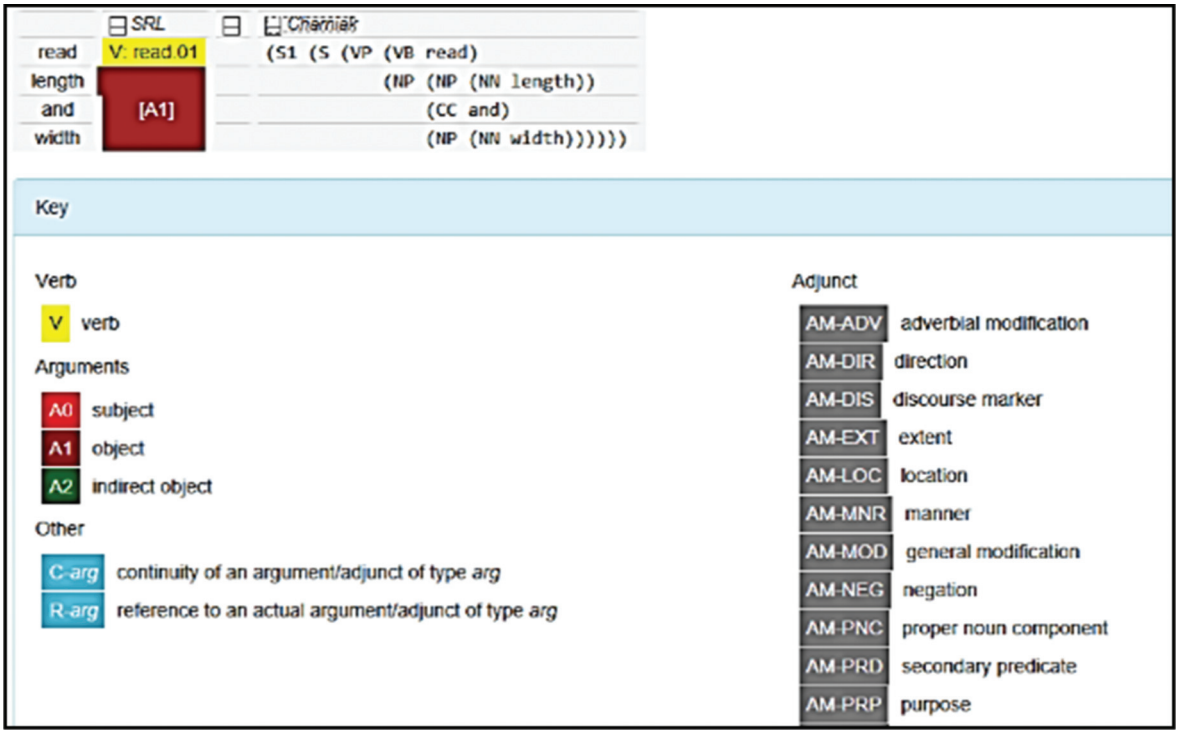


Figure 5: Linguistic Analysis Resulting from SRL Demo [41].

Word	Lexical Annotation	Semantic Annotation	Class of Verb
read	verb	-	???
length	noun	object	-
and	-	-	-
width	noun	object	-

Figure 6: Matrix Structure for Linguistic Analysis of the Information in a Pseudocode Statement.

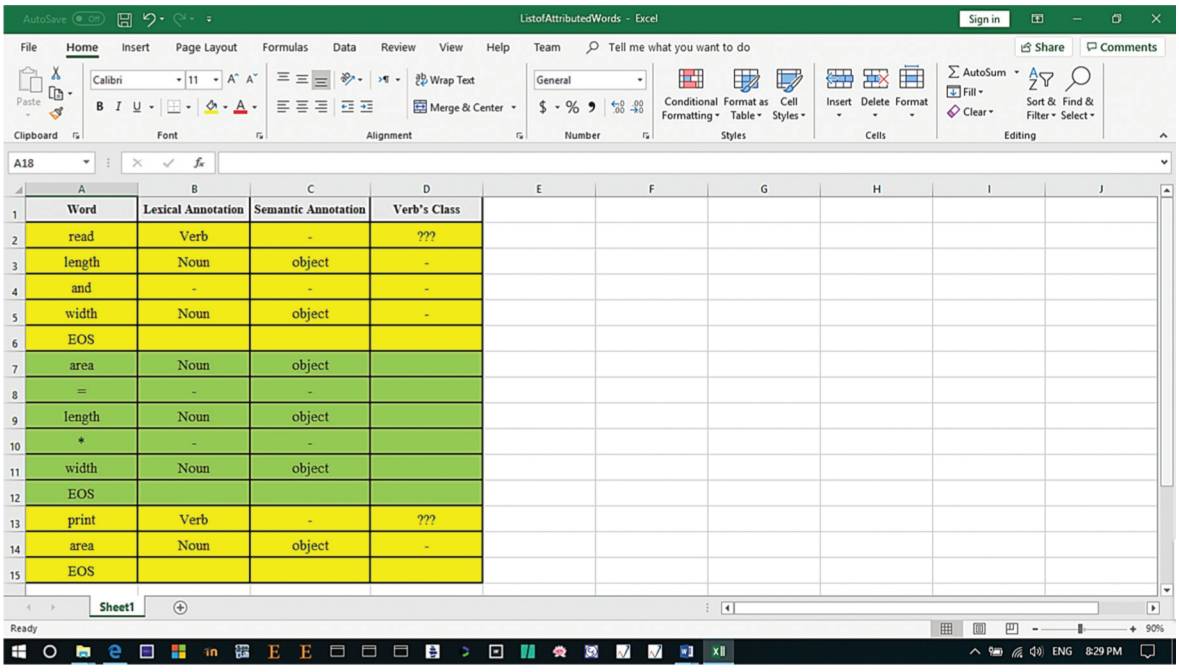


Figure 7: Excel File Showing the Structure of the Linguistic Matrix.

This matrix is filled in and saved in an Excel file manually. Figure 7 shows a screen shot of a file containing the "list of attributed words" of the pseudocode given above.

3.1.2 Identifying Class of Verb

From the corpora listed earlier, in this paper, we chose the VN corpus [45] as a tool for recognizing the semantics of a verb, treating the classes of verbs in VN as the semantics. This view is supported by previously implemented MT applications that were developed using Levin's classes of verbs [17]. In this step, the identification of the class of the verb is achieved by searching for the entry for a verb. This search is carried out by comparing the verb's attributes identified using SRL with the thematic roles and restrictions and the semantics fields for each verb entry in the library of the VN project. An example is given in Table 2, which shows a VN entry for class Hit-18.1. VN groups English language verbs using thematic roles and selectional restrictions for arguments, and defines frames as discussed below [45].

– *Thematic roles and restrictions*: Also called thematic roles or theta roles, they are assigned to each verb in a statement. Table 3 presents a representative set of semantic roles [18].
  Selectional restrictions are used to denote the general semantic boundaries that are imposed by a predicate (The predicate here is the verb.) on its defined arguments [25], where a predicate is "the part of a sentence that contains the verb and gives information about the subject" [38]. A failure to achieve compatibility

Table 2: Simplified VN Entry for the Hit-18.1 Class.

Class name	Hit-18.1
Thematic roles and restrictions	Agent[+int_control] Patient[+concrete] Instrument[+concrete]
Members	Bang, bash, hit, kick, ...
Frame's name	Basic transitive
Syntax	Agent V Patient
Semantics	Cause (agent, E) manner (during (E), directed motion, agent)! contact (during (E), agent, patient) manner (end (E), forceful, agent) contact (end (E), agent, patient)
Example	James hits the ball

Table 3: Table 3 Some Semantic/Thematic Roles.

Semantic role	Definition
Agent	Instigator of an event
Counter-agent	Force of resistance or against the action
Object	Entity involved in an action
Result	Entity resulting from an action
Instrument	Physical cause of an event
Source	Place something moves from
Goal	Place something moves to
Experiencer	Entity that accepts, receives, undergoes, or experiences the effect of an action
Actor	Agent's super type that controls, performs, instigates, or affects a predicate's situation

between these restrictions and the types of arguments leads to a semantic clash. Selectional restrictions play the role of a semantic grammar. Figure 8 shows the selectional restrictions associated with thematic roles [6].

- *Frame name*: This is used to describe the role of the verb. Examples are resultative, transitive, intransitive, and prepositional phrases [45].
- *Syntax*: This is a description that is used to recognize the structure of a verb's arguments and has importance in composition, including allowed prepositions.
- *Semantics*: This involves predicates of the restrictions that are used to impose thematic role types on the arguments of a verb and is also used to specify the possible syntactic nature of the arguments that are associated with these thematic roles. Examples of semantic restrictions are animate, human, and organizational. Conjunction predicates of Boolean semantics, such as "cause" or "contact", are associated with each frame for connecting it with other frames. An event variable denoted as E is also included in the predicate in order to specify when the predicate is true. Complete lists of the thematic roles, selectional and syntactic restrictions, and predicates are available on the Unified Verb Index Reference Page [44]. In this paper, a library of 328 verb entries is used, containing .XML files; this is offered by the VerbNet project, which is hosted at the University of Colorado Boulder [45]. Figure 9 shows the contents of an .XML file for a VN verb entry that is sought to find the class of the verb.

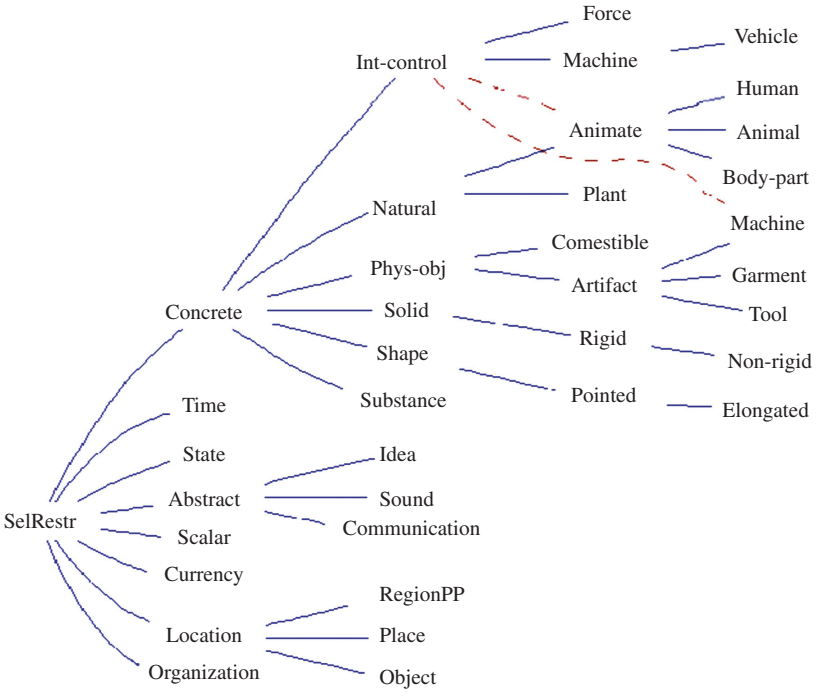
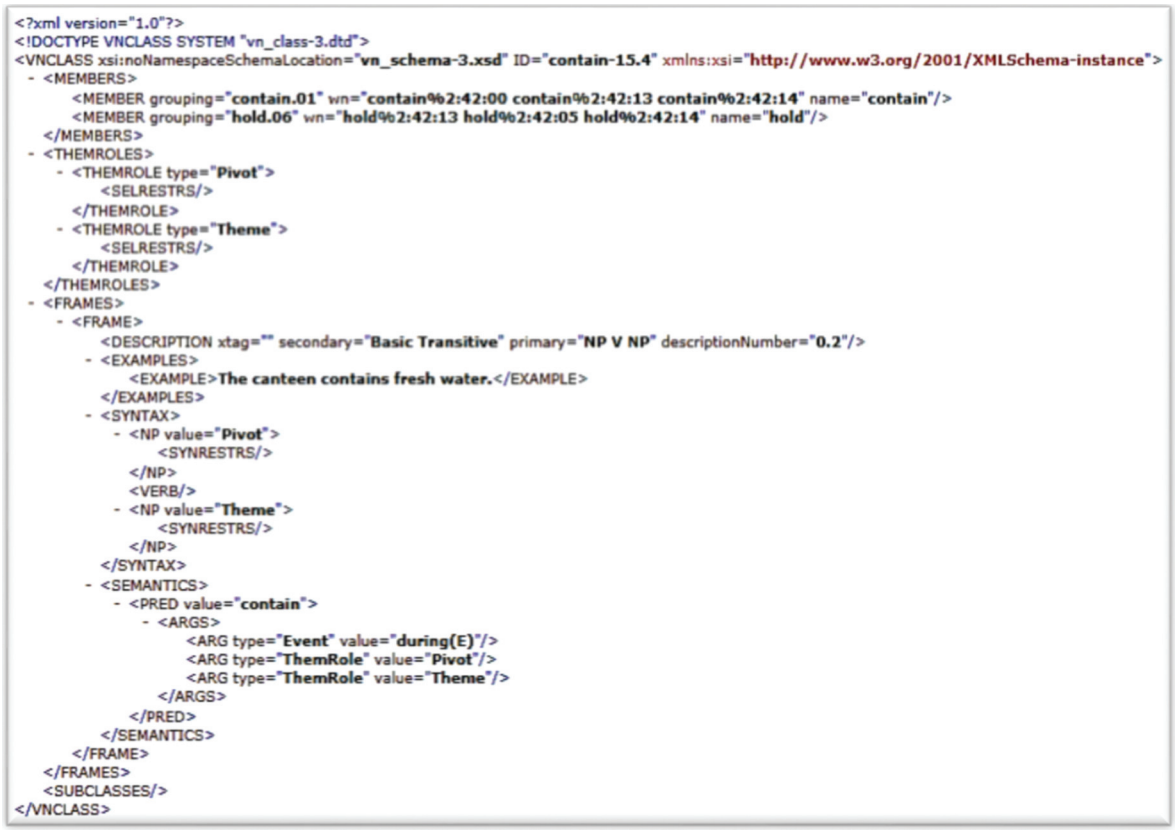


Figure 8: Selectional Restrictions Associated with Thematic Roles [44].



```
<?xml version="1.0"?>
<!DOCTYPE VNCLASS SYSTEM "vn_class-3.dtd">
<VNCLASS xsi:noNamespaceSchemaLocation="vn_schema-3.xsd" ID="contain-15.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 - <MEMBERS>
      <MEMBER grouping="contain.01" wn="contain%2:42:00 contain%2:42:13 contain%2:42:14" name="contain"/>
      <MEMBER grouping="hold.06" wn="hold%2:42:13 hold%2:42:05 hold%2:42:14" name="hold"/>
   </MEMBERS>
 - <THEMROLES>
    - <THEMROLE type="Pivot">
         <SELRESTRS/>
      </THEMROLE>
    - <THEMROLE type="Theme">
         <SELRESTRS/>
      </THEMROLE>
   </THEMROLES>
 - <FRAMES>
    - <FRAME>
         <DESCRIPTION xtag="" secondary="Basic Transitive" primary="NP V NP" descriptionNumber="0.2"/>
       - <EXAMPLES>
            <EXAMPLE>The canteen contains fresh water.</EXAMPLE>
         </EXAMPLES>
       - <SYNTAX>
          - <NP value="Pivot">
               <SYNRESTRS/>
            </NP>
            <VERB/>
          - <NP value="Theme">
               <SYNRESTRS/>
            </NP>
         </SYNTAX>
       - <SEMANTICS>
          - <PRED value="contain">
             - <ARGS>
                  <ARG type="Event" value="during(E)"/>
                  <ARG type="ThemRole" value="Pivot"/>
                  <ARG type="ThemRole" value="Theme"/>
               </ARGS>
            </PRED>
         </SEMANTICS>
      </FRAME>
   </FRAMES>
   <SUBCLASSES/>
</VNCLASS>
```

Figure 9: XML File for a Sample VN Verb Entry [45].

The file containing the linguistic analysis information for the pseudocode statements, which results from the linguistic analysis of the pseudocode statement, is read using the "Load" button in the CodeComposer program interface. Recall that the linguistic information is structured as a matrix, in which each row represents analysis information about a pseudocode statement. The contents of each row are used to guide the process of searching for the verb's class. This search is performed as a comparison between a row and each of the .XML files in the VN verb entry library. When a match is found, the class name for the verb entry is assigned to the cell in the row representing the class of the verb. The algorithm used to find the verb's class is given below:

```
String Function FindVerbClass (Row) (PRED, ArgType, and VNCLASS are names of tags in the XML file for the verb entry.)
Begin
While not end of VN_Verb_Entry_Library_Files
      Begin
      Read next file
      Get PRED tag value from the file
      Get ArgType.ThemRole.Value from the file
      If (PRED tag value = = Row.Word) and
      (ArgType.ThemRole.Value = = Row.SemanticAnnotation)
      Then begin
                Row.VerbClass = VNCLASS.ID
                Exit
                End
   End
End.
```

Recall here that the ultimate output of the *NLP step* (via its two sub-steps) is a frame (internal representation) that encompasses the verb class (obtained from the verb classification step) and its parameters (obtained using the SRL online software in the linguistic step), which will be used as input to the following step, which is the semantic rule-based mapping machine. Table 4 shows a typical structure for the class of a verb and its accompanying semantic roles.

The result of performing this step (using the data shown in Figure 7) is shown in Figure 10, in which the "verb's class" cell is assigned with a value that represents the class of the verb in its row.

3.2 Semantic Rule-Based Mapping Machine

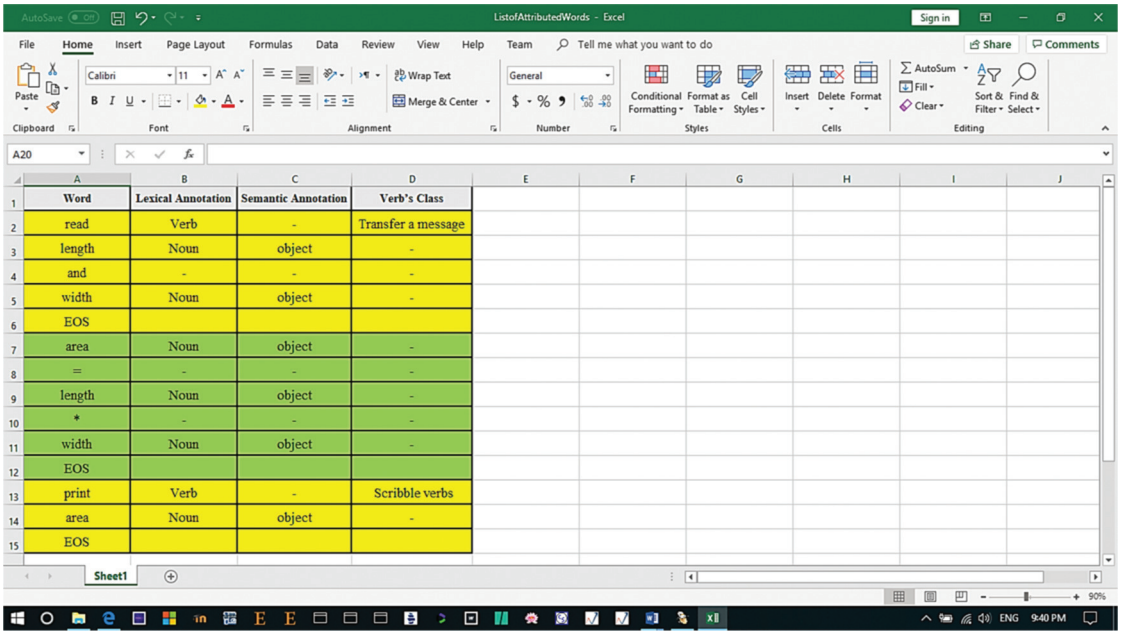
A logic-based approach used here to develop the mapping process is called a semantic rule-based mapping machine. The criteria or conditions used to govern the mapping process are the linguistic classes of verbs extracted in the step described in Section 3.1. These classes are used by a set of "if–then" production rules to map each pseudocode statement to its instruction in C#.

From a functionality point of view, the instructions of an algorithm (pseudocode) can be classified into three main categories: the first is the input statements, the second, the output statements, and the third, the processing statements [43]. The semantic role of each category is different from the semantic roles of the other categories. Table 5 lists the classes of pseudocode, their semantic roles, and the possible C# statements. The semantic rule-based mapping machine uses the resulting tokens and their attributes to identify a verb class and its related parameters.

Table 4: Class of a Verb and its Accompanying Semantic Roles.

Linguistic Verb Class	Accompanied Semantic Roles
Transfer a message	<object>*
Change of possession	<object><indirect object>
Alternating verbs	<object><indirect object>
Removing verbs	<object><source><indirect object>
Verbs for change of state	<object><object>
Say verbs	<manner><temporal>
Scribble verbs	<object>

*Means multiplicity, in that more than one object may exist with this verb class.



	A	B	C	D
1	Word	Lexical Annotation	Semantic Annotation	Verb's Class
2	read	Verb	-	Transfer a message
3	length	Noun	object	-
4	and	-	-	-
5	width	Noun	object	-
6	EOS			
7	area	Noun	object	-
8	=	-	-	-
9	length	Noun	object	-
10	*	-	-	-
11	width	Noun	object	-
12	EOS			
13	print	Verb	-	Scribble verbs
14	area	Noun	object	-
15	EOS			

Figure 10: Excel File Showing the Structure of Finding Verb's Class.

Table 5: Programming Statements Classes Versus Verb Classes.

Pseudocode instruction class	Verb class	C# statement/symbol
Input verbs (enter, input, read, get, ...)	Transfer a Message	Console.ReadLine( )
Processing instruction		
Assignment verbs (assign, move, set, initialize, store,...)	Change of possession	=
Arithmetic verbs		
– Add, append, total....	Alternating verbs	+
– Subtract, take, ...	Removing verbs	–
– Multiply	Verbs for change of state	*
– Divide	Verbs for change of state	/
– Calculate	Verbs for change of state	No opposite statement
Looping statements (repeat/until, do/while)	Say verbs	for (....)
Output verbs (write, print)	Scribble verbs	Console.WriteLine( )

The semantic rule-based mapping machine performs a composition task, in addition to selecting a suitable C# statement for the pseudocode statement. For example, the mapping for transferring a message verb class is performed as follows:

```
If (Row.VerbClass is “Transfer a Message”)
Then begin
    CSharpStatement= “ Console.ReadLine ( ”
    While (Row.Word<> “EOS”
        If (Row.SemanticAnnotation is “object”)
        Then CSharpStatement += Row.Word
        End if
    End while
    CSharpStatement += “);
End if
```

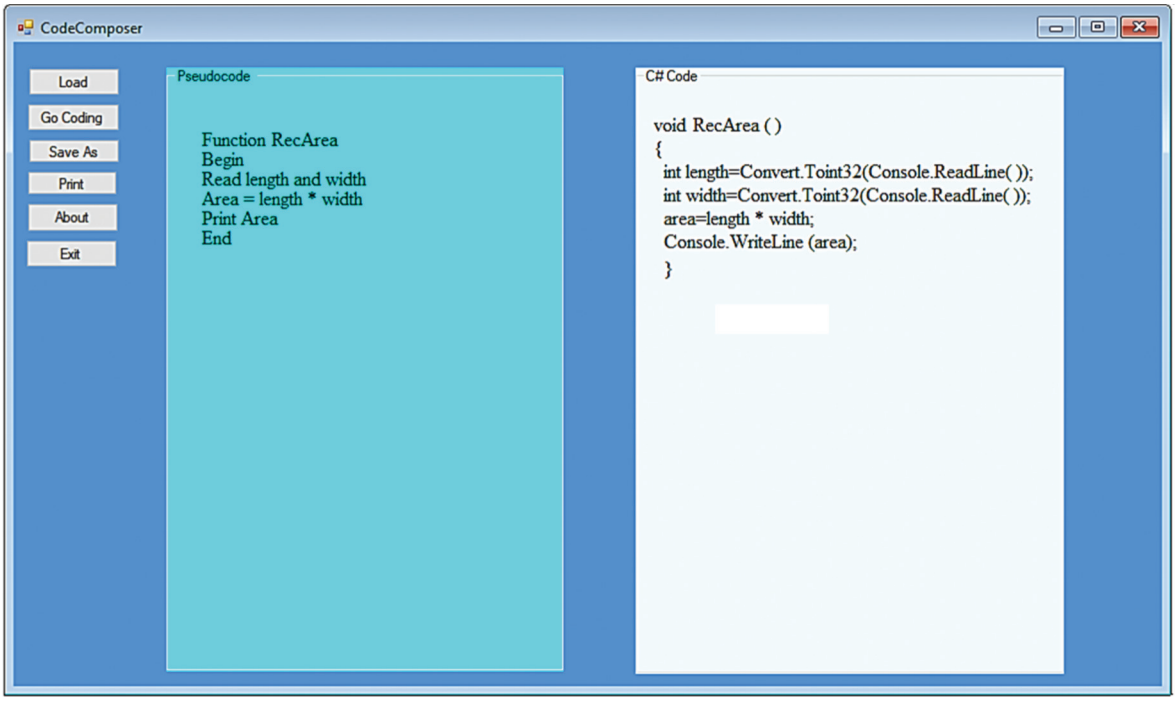


Figure 11: The Output of Testing a Pseudocode Using CodeComposer.

The output of this step is illustrated in Figure 11, and this also forms the ultimate output of CodeComposer. This output can be saved in a text file using the "Save As" button given in the GUI of CodeComposer. Although the pseudocode statements are submitted manually to SRL, which is the first step in CodeComposer, these pseudocode statements will be displayed in the interface of the CodeComposer program to give the reader a visual comparison between the pseudocode and the resulting C# programming language code.

4 Testing and Results

CodeComposer was run using multiple examples to test and demonstrate its performance. To calculate the binomial classification accuracy of CodeComposer, we use binomial classification accuracy [36], as follows:

– *Precision*: This is the proportion of actually translated items relative to the full number of items to be translated, and is calculated as

$$\text{Precision} = \text{TP}/(\text{TP} + \text{MsT}) \tag{1}$$

– *Recall*: This is the proportion of successfully retrieved items to the full number of demanded items, and is calculated as

$$\text{Recall} = TP/(TP + NoT) \tag{2}$$

– *F-measure*: This is the harmonic mean of recall and precision, and is calculated as

$$\text{F-measure} = (2 * \text{Precision} * \text{Recall})/(\text{Precision} + \text{Recall}) \tag{3}$$

where:

TP is the true positive, i.e. the number of translations that are free of errors
MsT is the mistranslated, i.e. the number of mistranslated pseudocode statements
NoT is the non-translated (NoT), i.e. the number of pseudocode statements that are not translated

Table 6 and Figure 12 show the results of running CodeComposer using 60 different forms of each of the pseudocode instruction classes illustrated in Table 5 as case studies. CodeComposer's results were evaluated

Table 6: CodeComposer Translation of Different Forms of Pseudocode Verb Classes.

Verb type	TP	MsT	NoT
Input verbs	53	5	2
Assignment verbs	50	7	3
Addition operation verbs	43	9	8
Subtraction operation verbs	49	7	4
Multiplication operation verbs	45	9	6
Division operation verbs	44	9	7
Looping statements	48	5	7
Output verbs	56	3	1

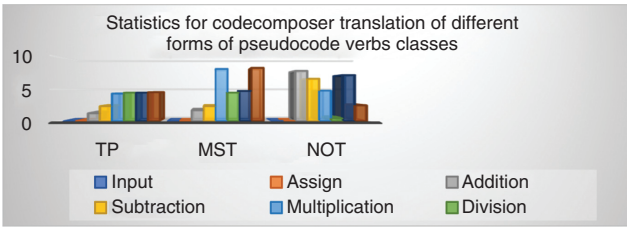


Figure 12: Effectiveness of CodeComposer.

manually to count TP, Mst, and NoT parameters. These results were used to evaluate the effectiveness of CodeComposer. The binomial classification accuracy of CodeComposer is reported in Table 7 and illustrated in Figure 13.

5 Discussion

The Related Works section of this paper focuses on the approaches used by related works, which are reported here in the form of examples of the processing approaches used. Templates, classes, generic frames, aspects, and prototypes are common ontological models used by generative programs and are integrated using processing tools such as template processors and pattern replacers that are governed by defined simple rules. A combination of these was used in an example of a source code generator [9]. Our proposed CodeComposer can be considered as a semi-automated approach that involves a combination of NLP software and an MT to perform the process of encoding a pseudocode.

The role of the theoretical concept of using NLP in the conversion of a pseudocode to C# code is crucial, as many of the words in the pseudocode are in natural English language and give rise to a great deal of complexity when automating the process of converting it to a programming language code. The use of a verb class is the most important part of this work, as the instructions in the pseudocode are verbs, which have no semantic roles as nouns, and hence need to be properly semantically mapped to their equivalent C# statements. The positive effect of this concept in developing CodeComposer is proven by its binomial classification accuracy.

The effectiveness of CodeComposer was measured using binomial classification accuracy, in terms of precision, recall, and F-measure. The overall precision of CodeComposer was 88%, which shows the wide linguistic area it covers. The recall of CodeComposer was 91%, indicating its strong ability to generate a valid translation. The F-measure of CodeComposer was 89%, demonstrating its accuracy.

In this work, we used the binomial classification accuracy method to evaluate CodeComposer instead of making comparison with the previous related works that are reported in this paper. This is because the comparison requires the use of the same data that were used by those systems, and obtaining such data is a very difficult thing. Therefore, we considered the manual converting of pseudocode to C# programming language

Table 7: Binomial Classification Accuracy of CodeComposer.

Verb type	Precision	Recall	F-measure
Input verbs	91%	96%	94%
Assignment verbs	88%	94%	91%
Addition operation verbs	83%	84%	83%
Subtraction operation verbs	88%	92%	90%
Multiplication operation verbs	83%	88%	86%
Division operation verbs	83%	86%	85%
Looping statements	91%	87%	89%
Output verbs	95%	98%	97%
Overall	88%	91%	89%

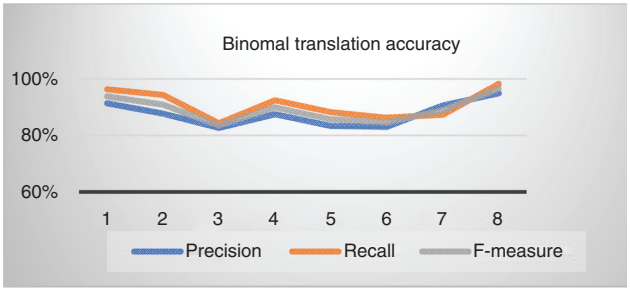


Figure 13: Binomial Classification Accuracy of CodeComposer.

Table 8: SRL Approach Used by CodeComposer Versus Other Approaches.

Other approach	Problem	CodeComposer
Text processing	Blind mapping	Semantic (intelligent) mapping
MDE	Requires developers to have significant abstraction capacity to develop MDE-based tools	Free of abstraction
MT	Hard to be used successfully with natural language like pseudocode	Works successfully with pseudocode (see binomial classification accuracy)
LBMT	Poor semantic processing	Based mainly on semantic processing
KBMT	The potential achievements of KBMT depend on the presence of a well-defined model of conceptual domains	Need no conceptual domain
EMBT	Has complexities to achieve its goal	Three steps to achieve its goal
Logic based	Consider no advanced topics of NLP (specially the semantic part) as a main requirement in the converting process	Count on the advanced topic of semantic processing verb classification, thematic role, and SRL
Deductive and inductive	Each one of these two approaches is used as a part of a process that automates the generation of a solution for a problem, as these two approaches do not convert pseudocode to programming code by their own	Convert pseudocode to programming code by its alone
NLP	Does not consider verb classification, thematic role, and SRL; the advanced topics of natural language semantic processing.	Count on the advanced topic of semantic processing: verb classification, thematic role, and SRL

as a benchmark that is used to evaluate the achievement of CodeComposer. The binomial classification accuracy technique follows this evaluation method to measure the accuracy of a system, which motivated us to use it in measuring the achievement of CodeComposer.

The simplicity of the mapping for the linguistically analyzed statements of the pseudocode justifies the use of if–then rules rather than other complex approaches such as ANN and GA that are used in related works.

Table 8 abbreviates a comparison between CodeComposer, which consider verb classification, thematic role, and SRL; the advanced topics of natural language semantic processing, and other approaches reported in Section 2 (Related Approaches and Works) of this paper.

As shown in Table 8, CodeComposer contributes the work on this field by using the advanced topic of semantic processing: verb classification, thematic role, and SRL. However, there are several criticisms of the NLP computation that should be noted. Aspects such as the use of slang, redundancy, multiple syllables, and complex ambiguity limit the success of development of such translation programs. The SRL software that is used to reveal the thematic roles of the tokens corresponding to the verbs still needs more work in terms of the number of thematic roles and the size of the corpus used. Tokens other than verbs pose no difficulties in terms of translation, as they contain less ambiguity and, hence, can be translated directly to their corresponding C# statements.

6 Conclusion and Future Work

Code generation by a computer extends the usefulness of an algorithm's design. Developers can benefit from the automatic conversion of their pseudocode into a programming language code that can be put to work immediately.

In this paper, we contribute methodologies for ACG through the use of a natural language translation approach, which includes a semantic rule-based machine for generating a C# code from a pseudocode. Our suggested approach to using a semantic rule to achieve translation from a pseudocode to a programming code is based on the fact that a pseudocode is a natural language that needs semantic processing to solve the typical ambiguity problems of natural language. A software tool of SRL is used to discover the meaning of an algorithm's instructions (linguistic verbs) using the semantic roles associated with the verb and, hence, to specify the verb's class, which in turn helps in the accurate mapping of the algorithm's instruction to the

correct programming statement. The VerbNet project library [45] was the cornerstone used in determining the class of the verb.

The CodeComposer software was developed using the approach described above, with C# as a language and .Net as an integrated environment; both of these offer a successful development environment for the implementation of CodeComposer as many of their properties support modern development requirements. CodeComposer is considered an I-CASE tool, as it utilizes NLP, an aspect of AI, in its work. Although there was some room for improvement, the results yielded by CodeComposer demonstrated the soundness and effectiveness of this approach in generating programming language codes. The resultant C# code for each example of input pseudocode shows the need for human revision to check the completeness of the translation from pseudocode to C# code. This is required due to several errors arising in the C# code, which may be of two types: mistranslated (MsT) and non-translated (NoT). An evaluation of CodeComposer was carried out using a binomial accuracy classification technique. We believe that this technique is realistic as it compares the output of CodeComposer to a manual output and, thus, describes the accuracy of CodeComposer relative to the accuracy of a human being in the conversion process.

It is important to mention that the key aspects of this work were the SRL tools. The shortcomings of the SRL software tools are quite clear in the dealing with the text that is missing good formation, such as slang, complex ambiguity, and redundant words. Therefore, we highly recommend the development of a more accurate and effective SRL software tool that can take on more semantic roles. Such a software tool would increase the quality of CodeComposer. Other recommendations for improving CodeComposer include both a comparative study of CodeComposer and other systems, and a feasibility study of CodeComposer in order to probe and evaluate its benefits and usefulness in the software industry. Another suggestion would be to re-design CodeComposer as a client–server application for commercial purposes.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi and J. Ullman, *Compilers principles, techniques, and tools*, 2nd ed., Addison Wesley, USA, 2008.

[2] F. B. K. B. Aïcha Choutri, A tile logic based approach for software architecture description analysis, *J. Softw. Eng. Appl.* **3** (2010), 1067–1079.

[3] Ajhais, *Converting algorithm to code*, Slashdot Media, NY, USA, 2013.

[4] M. R. Amal, C. V. Jamsheedh and L. S. Mathew, Pseudocode to source programming language translator, *IJCSITY* **4** (2016), 21–29.

[5] H. Ammar, W. Abdelmoez and M. S. Hamdi, Software engineering using artificial intelligence techniques: current state and open problems, in: *First Taibah University International Conference on Computing and Information Technology*, Al-Madinah Al-Munawwarah, Saudi Arabia, 2012.

[6] N. Asher, Selectional restrictions, types and categories, *J. Appl. Logic* **12** (2014), 75–78.

[7] E. C. M. Association, *C# language specifications*, Ecma International, Geneva, Swiss, 2006.

[8] C. F. Baker, C. J. Fillmore and J. B. Lowe, The Berkeley FrameNet Project, in: *The 17th International Conference on Computational Linguistics*, Montreal, Quebec, Canada, 1998.

[9] M. M. Baskaran, J. Ramanujam and P. Sadayappan, Automatic C-to-CUDA code generation for affine programs, in: *The 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, Paphos, Cyprus, 2010.

[10] K. Becker and J. Gottschlich, *AI programmer: autonomously creating software programs using genetic algorithms*, Cornell University, NY, USA, 2017.

[11] D. Bell, *Software engineering for students: a programming approach*, 4th ed., Longman Group, London, United Kingdom, 2005.

[12] D. C. Brown, Artificial intelligence for design process improvement, in: *Design Process Improvement: A Review of Current Practice*, pp. 158–173, Springer-Verlag London, London, UK, 2005.

[13] G. Canfora, A. Cimitile and U. D. Carlini, A logic-based approach to reverse engineering tools production, *IEEE Trans. Softw. Eng.* **18** (1992), 1053–1064.

[14] G. N. Carlson, A unified analysis of the English bare plural, *Linguist. Philos.* **1** (1977), 413–455.

[15] Y. Danilchenko and R. Fox, Automated code generation using case-based reasoning, routine design and template-based programming, in: *23rd Midwest Artificial Intelligence and Cognitive Science Conference*, Cincinnati, Ohio, USA, 2012.

[16] V. G. Díaz, E. N. Valdez, J. Espada, B. P. García-Bustelo, J. C. Lovelle and C. Marín, A brief introduction to model-driven engineering, *Tecnura J.* **18** (2014), 127–142.

[17] B. J. Dorr, Translation, large-scale dictionary construction for foreign language tutoring and interlingual machine, *Mach. Transl.* **12** (1997), 271–322.
[18] C. J. Fillmore, Types of lexical information, in: *Semantics: An Interdisciplinary Reader in Philosophy, Linguistics and Psychology*, pp. 370–392, Cambridge University Press, London, U.K., 1971.
[19] flowgorithm.org, "flowgorithm," 2017. [Online]. Available: http://www.flowgorithm.org/index.htm. [Accessed 15 10 2017].
[20] O. Furuse and H. Iida, An example based method for transfer driven MT, in: *4th International Conference on Theoretical and Methodological Issues in MT*, Montreal, Canada, 1992.
[21] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns elements of reusable object-oriented software*, 1st ed., Addison-Wesley Professional, USA, 1994.
[22] A. T. Imam, PAS2C: a new trend of translating program, *J. Educ. Sci.* **31** (1998), 91–96.
[23] A. T. Imam, S. Aljawarneh and T. Rousan, An expert code generator using rule-based and frames knowledge representation techniques, in: *5th International Conference on Information and Communication Systems (ICICS)*, Irbid, Jordan, 2014.
[24] A. T. Imam, A. J. Al-Nsour and A. Al-Hroob, The definition of intelligent computer aided software engineering (I-CASE) tools, *J. Inf. Eng. Appl.* **5** (2015), 47–56.
[25] R. Jackendoff, *Semantic structures*, MIT Press, Cambridge, MA and London, 1990.
[26] D. B. Jones, *Analogical natural language processing (studies in computational linguistics)*, UCL Press, London, UK, 1996.
[27] D. Jurafsky and J. H. Martin, *Speech and language processing*, vol. 3, Pearson London, London, 2014.
[28] P. Kingsbury and M. Palmer, From TreeBank to PropBank, in: *Third International Conference on Language Resources and Evaluation*, Las Palmas, Canary Islands, Spain, 2002.
[29] E. Kitzelmann, Inductive programming: a survey of program synthesis techniques, in: *Third International Workshop on Approaches and Applications of Inductive Programming*, Edinburgh, UK, 2009.
[30] J. Klein, H. Levinson and J. Marchetti, *Model-driven engineering: automatic code generation and beyond, Software Engineering*, Carnegie Mellon University, USA, 2015.
[31] D. Lonsdale, T. Mitamura and E. Nyberg, Acquisition of large lexicons for practical knowledge-based MT, *Mach. Transl.* **9** (1994), 251–283.
[32] G. F. Luger, *Artificial intelligence: structures and strategies for complex problem solving*, 6th ed., Pearson, London, UK, 2008.
[33] A. Meyers, R. Reeves, C. A. Macleod, R. Szekely, V. Zielinska, B. Young and R. Grishman, *The NomBank project: an interim report*, Association for Computational Linguistics, Massachusetts, USA, 2004.
[34] S. Mukherjee and T. Chakrabarti, Automatic algorithm specification to source code translation, *IJCSE* **2** (2011), 146–159.
[35] S. Nadkarni, P. Panchmatia, T. Karwa and S. Kurhade, Semi natural language algorithm to programming language interpreter, in: *2016 International Conference on Advances in Human Machine Interaction (HMI)*, Doddaballapur, India, 2016.
[36] D. L. Olson and D. Delen, *Advanced data mining techniques*, 1st ed., p. 38, Springer, Berlin, Germany, 2008.
[37] V. Parekh and D. Nilesh, Pseudocode to source code translation, *JETIR* **3** (2016), 47–52.
[38] C. U. Press, "Meaning of "predicate" in the English Dictionary," 2018. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/predicate.
[39] R. S. Pressman and B. R. Maxim, *Software engineering: a practitioner's approach*, 8/e, McGraw-Hill Global Education Holdings, LLC, NY, USA, 2015.
[40] V. Punyakanok, D. Roth and W.-T. Yih, The importance of syntactic parsing and inference in semantic role labeling, *Comput. Linguist.* **34** (2008), 257–287.
[41] V. Punyakanok, D. Roth and W. Yih, "Demo," 20 9 2017. [Online]. Available: http://cogcomp.org/page/demo_view/srl.
[42] M. Ramachandran, Automated improvement for component reuse, *Software Process: Improvement and Practice* **11** (2006), 591–599.
[43] D. S. Reddy, *Algorithm to code converter*, Weebly, Bharat Nagar, India, 2011.
[44] C. L. a. E. Research, "A Class-Based Verb Lexicon," University of Colorado, 11 8 2017. [Online]. Available: http://verbs.colorado.edu/~mpalmer/projects/verbnet.html.
[45] C. L. a. E. Research, "Unified Verb Index," University of Colorado, 15 8 2017. [Online]. Available: http://verbs.colorado.edu/verb-index/vn3.3/vn/class-h.php.
[46] S. Sato, CTM: an example-based translation aid system, in: *COLING '92 Proceedings of the 14th Conference on Computational Linguistics*, Nantes, France, 1992.
[47] K. K. Schuler, *Verbnet: a broad-coverage, comprehensive verb lexicon*, University of Pennsylvania, PA, USA, 2005.
[48] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Pearson Education, Inc, p. 4, Boston, MA, USA, 2011.
[49] J. Sejans and O. Nikiforova, Problems and perspectives of code generation from UML class diagram, *Scientific Journal of Riga Technical University. Computer Sciences* **47** (2011), 75–84.
[50] A. Software, "AthTek flowchart to code," 2017. [Online]. Available: http://www.athtek.com/flowchart-to-code.html#.WehMhY-CyM9. [Accessed 15 10 2017].
[51] I. Sommerville, *Software engineering*, 9th ed., Pearson, London, UK, 2010.

[52] L. M. Surhone, M. T. Tennoe and S. F. Henssonow, *Comparison of code generation tools*, Betascript Publishing, GmbH, 2010.
[53] I. TechTarget, "Pseudocode," 10 10 2017. [Online]. Available: http://whatis.techtarget.com/definition/pseudocode.
[54] J. Tsujii and S. Ananiadou, Knowledge based processing in MT, in: *The 1st International Conference on Knowledge Bases and Knowledge Sharing*, Tokyo, Japan, 1993.
[55] P. University, "What is WordNet?," 2015. [Online]. Available: http://wordnet.princeton.edu/wordnet/.
[56] Z. Zhang and Y. Zhou, A fuzzy logic based approach for software testing, *Int. J. Pattern Recognit. Artif. Intell.* **21** (2007), 709–722.