J Inf Process Syst, Vol.17, No.3, pp.537~555, June 2021
https://doi.org/10.3745/JIPS.04.0216

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

A Survey of Automatic Code Generation from Natural Language

Jiho Shin* and Jaechang Nam**

Abstract
Many researchers have carried out studies related to programming languages since the beginning of computer science. Besides programming with traditional programming languages (i.e., procedural, object-oriented, functional programming language, etc.), a new paradigm of programming is being carried out. It is programming with natural language. By programming with natural language, we expect that it will free our expressiveness in contrast to programming languages which have strong constraints in syntax. This paper surveys the approaches that generate source code automatically from a natural language description. We also categorize the approaches by their forms of input and output. Finally, we analyze the current trend of approaches and suggest the future direction of this research domain to improve automatic code generation with natural language. From the analysis, we state that researchers should work on customizing language models in the domain of source code and explore better representations of source code such as embedding techniques and pre-trained models which have been proved to work well on natural language processing tasks.

Keywords
Naturalistic Programming, Software Engineering, Survey, Source Code Generation

1. Introduction

The programming languages we used to develop software products have limitations. First, the cost of learning different programming languages is high for novice developers [1,2]. Second, software products are getting too complex, leading even expert developers to have a difficult time understanding code that others developed [3-5]. Last, programming language limits our expressiveness because we have to translate logical thinking into a foreign (programming) language [3,5].

These problems state that programming has high entry barriers. White and Sivitanides [6] presented a theory that programming languages require cognitive characteristics of formal operations, which is the highest cognitive development level. Unfortunately, many people fail to achieve formal operations. Ghezzi and Mandrioli [7] state that to be a software engineer, one must master the foundation, design methods, technology, and tools of the engineering discipline. They also claim that software engineers must have a solid background in all fundamental areas of mathematics such as continuous mathematics, discrete mathematics, and statistics/probability theory.

Programming with natural language might be one of the potential ways to mitigate these barriers

Manuscript received January 31, 2020; first revision March 23, 2020; accepted April 3, 2020.

Corresponding Author: Jaechang Nam (jcnam@handong.edu)
* Dept. of Information and Communication Technology, Handong Global University, Pohang, Korea (jihoshin@handong.edu)
** School of Computer Science and Electrical Engineering, Handong Global University, Pohang, Korea (jcnam@handong.edu)

although it may not directly resolve these challenges now. One of the main difficulties in programming is that it requires the ability to deal with abstractions and logical thinking, to form a hypothesis, to solve problems systematically, and to perform mental manipulations [6]. These skills have to be done within the syntax of a certain programming language. However, we use these skills quite “naturally” with our natural language. Thus, programming with natural language could be an initial step and an important direction to mitigate the entry barriers of programming.

In this sense, many researchers have studied code generation from natural language [1,8-12]. With a natural language description, the system generates a corresponding code snippet or fully runnable source code. For example, Pegasus [13], a tool that generates Java source code from the natural language description, receives a description of a program. With the received program description, it will produce fully runnable source code.

However, these studies have vague research goals and motivations because each proposed approach has studied without general and formal descriptions of the program with inputs and outputs. For instance, Pegasus [13], requires a program description of code in a line-by-line fashion to produce a corresponding source code as the output. Other approaches, such as Macho [14], take an abstract description of code and a test case example to supplement the ambiguity caused by the abstraction and produce the corresponding program. There are many approaches that generate source code from natural language, but their forms are very variant, and the goal of the research field is unclear.

The goal of this paper is to survey and review the approaches that generate source code with natural language descriptions. The first study in this research field [11] has started many years ago, but the follow-up studies have not been actively conducted because of the limitations of the technology. In the past decade, this research field got more attention since various deep learning and natural language processing (NLP) techniques [8,15-17] have been proposed. From surveying and analyzing the trend of this research field, we suggest that more studies related to deep neural architectures for source code generation should be explored.

2. Related Work

2.1 A Survey of Naturalistic Programming Technologies

Pulido-Prieto and Juarez-Martinez [5] listed 31 different approaches about tools and programming languages that assist users to program with the elements of natural language features. The elements of natural language, which they call naturalistic programming, is a formal and deterministic implementation of features from natural language. Examples of natural language features consist of deixis, expressiveness, phrases, anaphoric relations, context, ambiguity, etc. They tabulated tools by whether they provide the following functionalities: automatic code generation, reserved keywords, pre-defined grammars, data dictionaries, predefined code fragments, multiple programming language support, indirect references, industry-focused, learning-focused, documentation-focused. Second, they tabulated languages by the following functions: code generation, data dictionaries, English-like expressiveness, indirect references, learning-focused, and report generation.

The difference between their survey and ours is that (1) they have a broader concept of natural language because they consider certain features while we only consider actual natural language texts. (2) The approaches they list do not necessarily produce source code because they focus on assisting in programming, while our survey focuses on automatically generating source code.

2.2 A Survey of Machine Learning for Big Code and Naturalness

Allamanis et al. [3] list machine and deep learning approaches that learn about many aspects of different source code corpora. The authors categorized the models into three groups according to the form of different modalities they focus: (1) code-generating models, (2) representational models of code, and (3) pattern-mining models.

Code-generating models are probabilistic models that describe the corpus of source code in a stochastic process for generating source code. Representational models of code capture intermediate representations such as vector embedding. The pattern-mining models of source code focus on capturing human-interpretable patterns that can directly help software engineers.

The focus of our survey is classified as a code-generative multimodal model because receiving natural language description is considered receiving non-code modality for context. Our survey differs from that of Allamanis et al. [3] because they only consider techniques regards to machine or deep learning. Furthermore, the survey of them considers mostly on approaches that learn the representation of code. There are some approaches regards to generation of code but most of them do not focus on generating from natural language description of code [3]. However, our survey comprises both machine and non-machine learning techniques that generate source code from natural language.

3. Notations

This section specifies each notation used to categorize the approaches. We divided the notations into three parts: program form, input form, and output form. The taxonomy we organized uses these notations to show the different structures of each approach. The notations with upper-case letters indicate the modality of natural language while notations with lower-case letters indicate modalities of non-natural language. Natural language modality has the form of description. Modalities of non-natural language have the form of the actual value of notation or source code.

3.1 Program Form

A notation of a program form is an essential part of all the approaches. "$P_{L\ or\ A}$" is on the left-hand side of the structure, indicating that the approaches receive a natural language description of the program. On the right-hand side of the structure, "*c or s or m*" are produced, indicating that approaches generate a certain form of a program as output.

3.1.1 P_L - description of code (line-by-line)

"P_L" is a notation in the form of natural language where it describes a code in a line-by-line fashion. For example, "read an integer *i* and increment *i* until 100. While incrementing, add *i* to an integer *sum*. Print the integer *sum* on the screen." This is an example code that sums integer values from 0 to 100. It has a similar structure of pseudocode in the form of natural language. Due to its structural property, it lacks abstraction and makes the natural language description less "natural'.

3.1.2 P_A - description of code (abstract)

"P_A" is a notation in the form of natural language where it describes the source code more abstractly. The followings are examples: "a program that sums integers 0 to 100" or "read a file and split the content by commas." This form of code description is much more abstract and "natural" than notation "P_L," but it is more ambiguous than "P_L."

3.1.3 *c* - actual program code

"*c*" is an actual program code generated from a natural language description of a program. The code is complete and runnable.

3.1.4 *s* - code snippet

"*s*" is also in the form of actual source code similar to "*c*." The difference is that "*c*" is a full-runnable code while code snippet is just a part of code that is not directly runnable but has some functions that the user wants to perform or handles only a certain part of logic in a program.

3.1.5 *m* - intermediate program

"*m*" is an intermediate form of a program to mediate the two extreme languages: natural language and programming language. For instance, it can be in the form of knowledge representation for a better understanding (for both computer and human perspective); it can be in the form of a skeleton code where it helps users develop from code structure; it can be in the form of a regular expression.

3.2 Input Form

The input form is the notation that is read into the generated programs. Some approaches require input form, but it is not mandatory. Since it is an "input" form, it is on the left-hand side of the structure. In Section 4, categories with an input form are marked as a subscript of "*P*" (e.g., $P_{L\{i\}}$).

3.2.1 *i* - the actual input

"*i*" indicates an input form that corresponds to the actual input that is read into the target program. For instance, operand literals in a mathematical word problem-solving program; Excel file in an Excel formatting program; and a corpus of source code in a program synthesis domain, etc.

3.2.2 *t* - test case

"*t*" is an input form of an actual test case. We made this notation for disambiguation because test cases comprise both actual input and the corresponding oracle (expected output).

3.3 Output Form

The output form is a notation that is produced from running the generated programs. Some approaches always produce output while others do not. The notation of the output form is on the right-hand side of the structure. There is only one type of output form which is "*o*" - the actual output. Notation "*o*" indicates the actual output that is resulted from the generated program. In Section 4, categories with output form are marked as a subscript of "*c*" (e.g., $c_{\{o\}}$).

4. Category

In this section, we list the categories of automatic code generation approaches with their form of structures notated with code, input, and output forms described in Section 3.

4.1 $P_{L\{i\}} \Rightarrow c_{\{o\}}$

Approaches in this category require a natural language description that has the form of a line-by-line fashion and actual input values to be computed. What these approaches produce for the $P_{L\{i\}}$ is fully runnable source code with the actual output value, $c_{\{o\}}$.

However, they are only able to generate source code for specific domains such as Excel functions or mathematical problems. By focusing on the specific types of programs, the approaches in this category could generate complete runnable source code when specific types of inputs and program descriptions are given. Also, the ability to handle the abstraction of natural language description is constrained and has a form of a line-by-line fashion.

4.1.1 An interactive simulation programming which converses in English

Heidorn [11] proposed a natural language conversing system that takes a description of a program to generate its corresponding source code. The program belongs to a certain domain where programs answer specific questions. The system first checks the completeness of the description of the program. If the first stated program description is sufficient, the program will produce the corresponding problem-solving program with the answer. If the program description is insufficient or ambiguous, the system will ask questions to resolve the ambiguity. The program takes in both description of the program and the input values which mostly consist of number literals to be computed. The system continuously asks users questions until a sufficient amount of ambiguity is resolved to answer the question from the program description. The generated source code is in the form of GPSS (Gordon's Programmable Simulation System or General-Purpose Simulation System). GPSS is a general-purpose programming language for a discrete-time simulation developed by Geoffrey Gordon [18]. It is used to show process flow-oriented simulation such as simulating workflows in factories.

4.1.2 Translating keywords into executable code

Little and Miller [12] proposed a system that translates keyword commands to executable code in web and Microsoft Word. This approach generates fully runnable code from the natural language description with the resulted web page or a document file. For instance, when a user types in a keyword command "click search button," it is translated to the text "click(findButton("search"))." The translated source code is then executed in the form of the web as an output. They implemented a similar function in the domain of Microsoft Word. For example, when a user types in "left margin 2 inches" the corresponding Visual Basic code "ActiveDocumen-t.PageSetup.LeftMargin = InchesToPoints(2)" will be generated with the document file.

4.1.3 NLyze: interactive programming by natural language for spreadsheet data

Gulwani and Marron [19] proposed a natural language-based interface for spreadsheet programming. This approach translates a natural language description of Excel functions such as algebraic calculations and table configurations to generate and rank corresponding program candidates. Aside from the description of an Excel function, NLyze takes in an input spreadsheet file as a source. As the outcome, the spreadsheet program candidates and the modified spreadsheet is generated.

4.1.4 Automatically solving number word problems by semantic parsing and reasoning

Shi et al. [20] proposed a system that uses semantic parsing and reasoning to generate source code that calculates mathematical word problems. The system receives a description of a mathematical word problem with the input numbers. Then, the system generates the corresponding source code and the answer to the math problem. The generated source code is in the form of the DOL language (DOlphin Language) which is a semantic representation language designed by Shi et al. [20].

4.1.5 Automatic code generation towards solving addition-subtraction word problems

Mandal and Naskar [21] proposed a system that generates a program solving mathematical word problems from natural language descriptions. The approach extracts relevant information from the natural language description and stores them into an object-oriented template. The mapped template is used to perform mathematical operations to solve the problems. For information extraction, they used natural language semantic role labeling and made a template called the OIA triplet (Owner-Item-Attribute) and stored the template with the extracted information. The input and output form of the system is identical to the approach in Section 4.1.4.

4.2 $P_L \Rightarrow c$

Approaches in this category take in program descriptions in the form of a line-by-line fashion to produce the corresponding source code. Some approaches take in actual values of input or produce an output, but they fall into this category because they do not always require them compared to the category $P_{L\{i\}} \Rightarrow c_{\{o\}}$.

Many approaches in this category are also capable of generating fully runnable source code and they can generate in a general-purpose language [1,2,13]. However, most of them are restricted in their "naturalness" of the input program description. Many approaches were in the early phase of this research field. They use techniques such as semantic parsing and information extraction. The reason that the input natural language descriptions were restricted is that these early techniques cannot fully capture the knowledge of natural language modalities.

4.2.1 NaturalJava: a natural language interface for programming in Java

Price et al. [1] proposed an interface for creating Java programs with line-by-line natural language descriptions of a program. The system is composed of three distinct subsystems to produce Java code from natural language descriptions. First, they use a NLP system called Sundance, which takes in a natural language description of the program to extract information. With the extracted information, they generate case frames that represent the essentials of the program. Case frames are a syntactic representation of sentences and pattern-based templates used in Sundance. Second, the case frames are passed to a subsystem called PRISM, which is a knowledge-based case frame interpreter that interprets case frames to generate program abstract syntax tree (AST) of Java. Lastly, the generated ASTs are passed to a Java AST manager called TreeFace, which translates the ASTs to an actual Java source code.

4.2.2 Framework for creating natural language UI for action-based applications

Chong and Pucella [22] proposed a framework that creates interfaces of action-based applications with natural language descriptions. The natural language descriptions are mostly interactions of users and the system. Their main component of this approach uses type-logical grammar to translate the natural language description of a program into a higher-logical expression. The resulted logical expressions are then passed to the action interpreter, which executes the corresponding action calls on the target application.

4.2.3 Spoken programs (Spoken Java)

Begel and Graham [23] proposed a voice programming interface that receives spelling word (speech), natural language, or paraphrasing text, to produce the corresponding Java program. This approach has especially dealt with ambiguity in the context of speech recognition such as homophones. However, ambiguity still resides in the point of written natural language.

4.2.4 Programming with unrestricted natural language

Vadas and Curran [2] proposed a natural language interface for programming. With unrestricted syntax, they used wide-coverage syntactic and semantic methods to extract information from the natural language description. It uses a combinatorial grammar parser to get the syntax of the natural language description. Although the input may take in unrestricted forms of description, the translation is done in a line-by-line fashion. For evaluation, they did a study of how people gave programming instructions in natural language. They found that most people preferred using programming terms than those of simple natural language.

4.2.5 Pegasus - first steps toward a naturalistic programming language

Knoll and Mezini [13] proposed a programming language that reads structured line-by-line natural language (in English, German, and Arabic [24]) and produces the corresponding program in Java. The basic features of Pegasus consist of reading natural language, generating source code, and expressing natural language. When reading natural language, it extracts keywords that have logical meaning such as *if* or *then*. With these keywords, it can extract the location of each statement and command clause. These pieces of information are then stored in a storage, which the authors define as *the brain*, in the form of an idea notation. The idea notation is a data structure defined to keep the syntax and the semantics of the natural language description. For the generation of source code, Pegasus uses the meaning-library, which is a database defined to match an idea notation and the corresponding Java commands. Pegasus can also express the program in natural language. Since the input programs are stored in *the brain* in the form of the idea notation, the system can backtrack to input natural language descriptions from the idea notation they have stored.

4.2.6 Natural language programming of a multiplayer online game

Leiberman and Ahmad [25] proposed MOOIDE for creating MOO with a natural language description of the program. MOO is a text-based, multiplayer, online, virtual reality system. MOOIDE analyzes the natural language description using natural language parser to capture information. It also has an interacting mechanism to add new elements that the user wants to elaborate in the simulated world. The natural language parser uses anaphora resolution and common-sense knowledge to guarantee that objects behave as intended. The user can simulate or create virtual space using MOOIDE with typing in program descriptions.

4.2.7 SmartSynth: synthesizing smartphone automation scripts from natural language

Le et al. [26] proposed a smartphone script generation tool that processes natural language description of a smartphone program. The system is designed to program on smartphones with various platforms using natural language. The synthesis algorithm uses NLP to identify different components and their dataflow relations. It also uses type-based program synthesis to infer other missing dataflow and generates the script from reverse parsing. While it processes the description, SmartSynth interacts with the user to resolve the ambiguity or unknown elements not specified in the input description.

4.2.8 NLCI: a natural language command interpreter

Landhaußer et al. [27] proposed a natural language command interpreter that takes in natural language description of a program and produces source code relevant API calls based on the ontology. The construction of the ontology can be automated if APIs use descriptive names for their components. NLCI is a domain agnostic because the domain knowledge can be changed and fine-tuned by changing the ontology before code generation. To check this attribute, they tested their system on two very different domains. First, they tested on Alice which is a 3D tool designed to teach children to create animation programs. Second, they tested on openHAB which is an API for home automation.

4.3 $P_A \Rightarrow s$

Approaches in this category take in program description that is more abstract and "natural." The generated source code, however, is in the form of a code snippet, rather than fully runnable source code contrast to the categories that take P_L as Categories 4.1 and 4.2.

Also, they can generate source code in general-purpose languages [8,10,15,17]. However, these approaches cannot generate fully runnable programs as ones from the approaches in Categories 4.1 and 4.2. Because many of these approaches use machine learning techniques to generate source codes [8,10,15-17]. These techniques suffer in generating sound and complete code due to *loss* and *error*. While natural language modalities are very robust in these *losses*, source codes in programming languages are very much affected by them. Still, this field of study is promising as quality and quantity data are piling up in software repositories.

4.3.1 Bimodal modeling of source code and natural language

Allamanis et al. [8] proposed bimodal modeling of natural language and source code. This literature is the first generative model to apply a neural language model on both source code and natural language.

The model uses aggregated datasets of natural language and source code. They experimented on using additive representation and the element-wise multiplicative representation in aggregating the datasets. They proved that element-wise multiplicative representation of data works better than the additive data. With the aggregated data, they used a neural language model called the log-bilinear model to train the aggregated data. Since their model is bimodal, they can produce source code from natural language descriptions, or they can generate natural language descriptions from source code. Their results indicate that generating source code from natural language is much more difficult than generating natural language from source code. This is because the generation of natural language is a much more robust task while the generation of source code consists of many obstacles such as considering the strict syntax of a programming language. The target programming language they used for the dataset was C#. They evaluated data from Stack Overflow, Dot Net Perls and have achieved 0.26 MMR average.

4.3.2 Synthesizing Java expressions from free-form queries

Gvero and Kuncak [28] proposed a code assistance tool for developing Java code. The system takes in a free-form query that can contain both natural language and code modalities to produce candidates of corresponding Java code expressions. The tool is composed of the following subsystems: (1) a customized NLP tool for information extraction, (2) a matching algorithm for connecting queries with code ingredients, (3) probabilistic context-free grammar (PCFG) models trained with Java corpus, and (4) an algorithm to generate Java expressions using the artifacts produced from other subsystems.

4.3.3 Learning semantic parsers for IFTTT (if-this-then-that) recipes

Quirk et al. [29] proposed a semantic parser that maps natural language description to an IFTTT recipe. IFTTT is widely used for web services to create chains of simple conditional statements. These chains trigger events for controlling web applications such as Gmail, Facebook, and Instagram. They trained a log-linear model in character-level with the n-gram features. They used a large corpus of IFTTT recipes aggregated with their natural language descriptions.

4.3.4 SWIM: synthesizing what I mean

Rahothaman et al. [30] proposed a system that translates free-form user queries into the APIs of interest and their corresponding code snippet. First, the natural language query is mapped to the API of interest. Second, the system retrieves a structured call sequence and its usage patterns of the target API. Last, they generate idiomatic code snippets from the structured call sequence. The query does not have to contain API specific keywords to generate the idiomatic snippet.

4.3.5 Latent predictor networks for code generation

Ling et al. [16] proposed a neural architecture called Latent Predictor Networks that marginalizes multiple predictors for efficient training and scalable generation of source code. The advantage of marginalization is that it can choose different contexts for training and the granularity of generated code. They used source code and natural language data from two card games *Magic the Gathering* and *Hearthstone* and Django. They also integrated an attention method to handle structured input sequences. The resulted BLEU and accuracy scores averaged 68.7 and 24.4, respectively.

4.3.6 Program synthesis using natural language

Desai et al. [10] proposed a program synthesizing framework that takes in a natural language description of a program and a training dataset to generate the corresponding source code. The training dataset consists of text-code pairs of a domain-specific language and the corresponding description. With the input dataset, the system learns the language and generates a code snippet that matches the description of the program. The system learns any language from the input dataset enabling the system to be language agnostic. If the user can provide any programmable code and text pairs, the system can generate the corresponding program of code. They evaluated their method on the Air Travel Information System (ATIS), Automata Theory Tutoring, and Repetitive Text Editing and correctly translated in top3 ranks with the average percentage of 91.1%.

4.3.7 A syntactic neural model for general-purpose code generations

Yin and Neubig [17] proposed a method to parse natural language descriptions to generate Python code snippets. They modeled neural architecture that uses a probabilistic grammar model to explicitly capture the syntax of the programming language as prior knowledge. They also found that this approach is effective in scalability when generating complex programs. They stated that this approach outperformed many code generations approaches that use semantic parsing. They evaluated *Hearthstone*, Django, and IFTTT and resulted in BLEU and accuracy score averaging 80.2 and 43.9, respectively.

4.3.8 Natural language to shell commands

Lin et al. [31,32] proposed a system that translates a natural language description to generate the corresponding shell command. The translation is done by using recurrent neural networks (RNNs) and semantic parsing. When the input natural language description is read, the descriptions are preprocessed with semantic parsing technique called named entity recognition (NER) to capture the details of the tokens. Then, the tokens are read into the RNN encoder-decoder model that is used to translate a natural language template to a program template. They used the nearest neighbor clustering to evaluate the compatibility of an entity that is generated. The measured compatibility is used to cluster commands in each group and the arguments are filled accordingly to the generated commands.

4.3.9 Seq2SQL: structured queries from natural language using reinforcement learning

Zhong et al. [33] proposed a model that generates a structured query from a natural language using reinforcement learning. The natural language description, in the form of questions, is translated to corresponding SQL queries. The model uses the mixed objective of reward weights and cross-entropy loss to train executions of the database and learn policies to generate conditions of SQL. This approach leverages the structure of SQL to limit the range of generated queries to simplify the generation problem. They experimented on WikiSQL and averaged 54.5 in accuracy scores.

4.3.10 Augmenting and structuring user queries for free-form code search (COCABU)

Sirres et al. [34] proposed a free-form search engine that resolves the vocabulary mismatch problem. With natural language or Java expressions, they augment the query to resolve the vocabulary mismatch problem and finds code examples that have high relevance from software repositories such as GitHub and Stack Overflow. This approach does not generate source code, but it retrieves code snippets that are fully functional because it finds code snippets from software repositories. The downside, however, is that it cannot produce or generate source code that is new or not existing in the software mentioned software repositories.

4.3.11 Deep code search

Gu et al. [15] proposed a deep neural network and a code search tool that takes in a natural language description of a snippet for retrieving the corresponding code snippet. The deep neural network, CODEnn (COde Description Embedding Neural Network), is trained to capture the semantic similarities of natural language description and the code snippet. The two different modalities are trained and embedded into unified vectors. When a code snippet and a description are semantically similar, the embedded vectors will be close to each other. With this model, they implemented a deep learning-based code search tool, DeepCS, and evaluated their approach. DeepCS recommends top K most relevant code snippets from a natural language description. They evaluated on Java corpus of 18M methods from GitHub and overall, they averaged MRR of 0.60.

4.3.12 Vajra: step-by-step programming with natural language

Schlegel et al. [35] proposed an end-user programming paradigm for Python. Vajra takes natural language descriptions and generates corresponding Python code snippets. The user types in natural language command to a specific spot in source code. Then, their system generates a list of possible statements and their associated parameters that are most similar in semantics. There are procedures that the user can choose to resolve ambiguity in the process by clicking from multiple candidate snippets. The core technique used in this study is semantic parsing.

4.4 $P_A \Rightarrow m$

Approaches in this category take an abstract natural language description of a program to generate a corresponding intermediate program form. As stated in Section 3, intermediate code needs secondary work to be processed before execution, i.e. implementation for a skeleton program, a compilation for medium (assembly) language, and code integration for a regular expression.

These approaches differ from other categories in that they generate what we define as intermediate code. They are not necessarily in the form of source code but if they are, they are in the form of abstraction of source code, i.e., a skeleton code that is not runnable. We added these approaches in this category because, even though they are not in the form of programming language or fully runnable codes, they receive natural language modalities and generates an output of a source code modalities that helps developers develop programs.

4.4.1 Metafor: visualizing stories as code

Lie and Lieberman [36] proposed a program editor that uses descriptive statements about a program to create scaffolding code fragments that can be used for the designers and developers. The system interacts with the user and uses the dialogue information for disambiguation. The system captures different objects, functions, and descriptions and makes those pieces of information as class abstraction and generates a skeleton program code in python language.

4.4.2 CPL & CPL-lite

Clark et al. [37,38] proposed CPL which stands for Computer-Processable Language. CPL receives natural language descriptions and generates an intermediate representation that restricts the descriptions to a subset of natural language so that both humans and computers can understand better than the two extreme languages, i.e., programming and natural languages. CPL uses heuristics to resolve the ambiguity in the natural language description. They have three types of sentence input: facts, questions, and rules. In CPL-lite, they added a mechanism to define queries in a comprehensive and controllable way. CPL-lite does not use heuristics but a more restricted interpreter to handle the ambiguity. The form that is generated is a program called knowledge machine (KM) which they proposed in [12]. KM is a mature, advanced, frame-based language with well-defined semantics, used previously in several major knowledge representation projects.

4.4.3 Automatic programming with natural language compiler

Somasundaram and Swaminathan [39] proposed a compiler that parses the natural language description of a program to generate intermediate representation to help the compiler to convert them into the target language with minimal effort. They aim to reduce ambiguity by parsing through the natural language descriptions of a problem statement and generate the corresponding object-oriented program. The key component of their approach consists of a syntactic analyzer, symbol table, lexical analyzer, semantic analyzer, intermediate code generator, and a code generator.

4.4.4 Assisted behavior-driven development using NLP

Soeken et al. [40] proposed an assisted flow for behavior driven development (BDD) where the user provides an acceptance test composed of natural language dialogue with the computer about code pieces. The system extracts code information using NLP techniques from the dialogue to produce skeleton code.

4.4.5 Using semantic unification to regular expression from natural language

Kushman and Barzilay [41] proposed a translator that takes free-form queries and performs a semantic unification to generate the corresponding regular expression. The introduced ambiguity from the two modalities differs, but regular expression also has multiple representations of the equivalent expressions. The author exploits this flexibility to facilitate translation by finding a form that is more similar to the natural language. They evaluated their technique on a set of natural language queries and their corresponding regular expressions gathered from Amazon Mechanical Turk [42].

4.5 $P_{A\{t\}} \Rightarrow c$

Approaches in this category receive an abstract natural language description of code and a set of unit tests (example inputs and their oracles), to generate fully runnable code.

They can handle abstract or “natural” description and requires unit tests as a requirement can improve the ambiguity from the “naturalness” of the description. Using unit tests enables them to handle abstract descriptions and generate fully runnable source code in a general-purpose language.

4.5.1 Macho: Programming with man pages

Cozzie and his colleagues [14,43] proposed a tool that generates source code from receiving abstract natural language and a unit test that has one or more examples with correct input and oracle of the program. In an abstract natural language description, there is always the challenge to resolve the ambiguity. Macho first parses the abstract natural language and create multiple candidate programs due to the ambiguity. Then, it checks the candidates with the unit test to see which fits the best. When presenting the candidate programs, Macho uses a raking system by using a probabilistic model. The project was trained on a large database of open-source Java codes.

4.5.2 Integrating programming by example and natural language programming

Manshadi et al. [44] proposed a system that generates source code from a natural language description. The system receives unconstrained instructions and one or more input and output examples. This approach differs from the others because they use the technique of version space algebra [45]. This technique is used to decompose a problem into simpler problems that can be individually solved to reduce the complexity of problem-solving. They also use the method of probabilistic programming by example (PPbE) to reduce the number of possible solutions.

5. Discussion

It is clear that code generation from natural language has potential and is promising in software engineering practice. The followings are our technical and trend analysis, current challenges, and future directions after we have organized the survey.

5.1 Technical Analysis

Approaches in category $P_{L\{i\}} \Rightarrow c_{\{o\}}$ are implemented by parsing the natural language description and translating the description through hard-coded grammar to generate fully runnable source code. This could be done by hard coding because the number of possible grammars is limited by the “naturalness” of the description and the domain of the program. The domain of the program is controlled by limiting the input and the output of the program. By limiting the two axes (“naturalness” and domain) in the search space, they were able to implement the grammar that generates fully runnable source code.

Approaches in category $P_L \Rightarrow c$ is also implemented by parsing natural language descriptions. However, they expanded the axis of the domain-specific program to generate a general-purpose program [1,2,13,40,44]. Due to the expansion of the search space, they implemented additional disambiguating grammars to correctly map the natural language description with the correct source code.

Approaches in category $P_A \Rightarrow s$ expands the search space in both axes to achieve “naturalness” and general-purpose program [8,10,15,17]. Approaches that can handle abstract natural language to generate general-purpose language use probabilistic language models or machine/deep learning models [8,15-17]. Due to the techniques’ inevitable loss and error, the generated source code’s completeness is reduced to the snippet level. Despite reducing the space of completeness, the approach’s performance is still very low.

Approaches in category $P_A \Rightarrow m$ is implemented by parsing the abstract natural language description and generates an intermediate level of source code [37-39]. Some approaches reduce the axis by generating domain-specific language such as regular expression [41]. Other approaches generate a general programming language of a skeleton program [36,40].

Approaches in category $P_{A\{t\}} \Rightarrow c$ is implemented by parsing the abstract natural language description to generate a complete general-purpose program. Even though the search space is large, the approaches effectively find the target source code by exploiting the provided oracles. However, providing the right number of effective oracles is difficult and cannot be done by the system.

5.2 Trend Analysis and Current Challenges

From surveying the approaches that generate source code from natural language, we could see the current research trends and challenges in this research field.

Former approaches in this field are implemented with techniques such as semantic parsing to find and extract information from the natural language descriptions. With the information extracted, they corresponded to them with certain codes with similar keywords. By using such techniques, the code generation was done in a line-by-line translating fashion. This means that every specific detail of the source code has to be given in the description. This weakens the abstraction and 'naturalness' of program descriptions. These aspects can be observed from categories such as $P_{L\{i\}} \Rightarrow c_{\{o\}}$, and $P_L \Rightarrow c$.

More recent approaches tried to resolve the restrictions of abstraction and 'naturalness' by using neural networks and probabilistic language models [8,10,15-17]. With abundant source code corpora in software repositories, these techniques have been studied actively. By using machine learning techniques, it resolved the restriction of handling "natural" program descriptions. It also made possible to generate source code even if the details of the program description are missing. It was possible to infer from the abstract description of the program. However, due to the inevitable *loss* and *error* that arouse in machine learning techniques, it introduced new challenges. With the *loss* and *error*, the generated source code was syntactically unsound. To improve the soundness of the generated code, researchers narrowed down the range of code to be generated, from completely runnable code to code snippet. However, the performance of generating sound source code is still an open challenge [8,15-17]. Researchers in this area have to work on building models that are more sound and complete for practicality. Another point is that these models are originally built for natural language. Although programming language and natural language have similar characteristics, the difference still does exist.

From looking at the characteristics of each category and their release dates, we could see the current trend of this research field. The approaches started from handling descriptions with a low level of abstraction (P_L) to a higher abstraction of natural language descriptions with supplementary information ($P_{A\{t\}}$) then finally to approaches with high abstraction without additional input (P_A).

5.3 Future Directions

As stated in Section 5.2, the current direction of this research field is going from handling descriptions with a low level of abstraction to handling descriptions with higher abstraction. From handling the abstract descriptions, we have lost the completeness of generated source code. To ensure the usability of code generation with natural language approaches, we need to enhance the ability to generate complete source code. To do this, we suggest the important direction as follows.

More "source code" focused probabilistic language and neural network models should be exploited.

Current approaches use models for processing natural language. There are distinct features that programming language have, but natural languages do not. The features are mostly very formal. Using the formality of programming language as specialized features as an objective function will help achieve better performance. By fine-tuning these techniques to source code domain, it will help the model fit within the correct syntax. Thus, it will be able to generate a much more complete source code. Another point is that these techniques receive a very abstract form of descriptions, having much more ambiguous descriptions. Ambiguity resolution is more crucial in the approaches that use probabilistic language models than approaches using semantic parsing techniques, but they never tend to do so. Approaches with P_L have focused on many disambiguating technologies that have a 'natural' interface such as asking questions to users. The ambiguity resolving techniques exploited in the approaches with P_L should be applied to probabilistic models for disambiguation to generate more complete program code. Another practical improvement can be made by studying the better representation of source code. The state-of-the-art benchmarks of NLP techniques show that fine-tuning deep pre-trained models improve most of the tasks [46-48]. However, exploiting the representation of large pre-trained models on source code corpora is currently unknown.

For future work, we are interested in applying machine and deep learning models directly on source code corpora to learn their features and find supporting knowledge to assist in software engineering practice. One application could be applying neural network models on Javadoc and their related code to generate source code from natural language program descriptions. We need to learn and capture more raw knowledge of source code corpora to build complete generative models of source code. A second application could be using an issue report in the form of natural language to automatically generate patch source code. Currently, patch generation involves intermediate processes such as generating test cases and applying them to generate patches. By predicting patches from the issue report, we can evade these intermediate processes. Another topic for future work will be integrating a conversation mechanism to probabilistic language models. When the natural language description is too broad or ambiguous, the system will ask users to elaborate on the description for disambiguation. Conversing programs, shown from categories with P_L works well as a disambiguation technique and have the most "natural" way of working. After capturing the knowledge of different features of source code and natural language and handling ambiguity, there will come a time when developers can program better from natural language. In a very distant future, we could communicate with computers by our natural languages as the famous *Jarvis* system introduced in the movie, *Iron Man*.

6. Conclusion

In this paper, we have surveyed and reviewed the different approaches of automatic code generation with natural language descriptions. After listing the approaches, we categorized them by their structure and analyzed the technical issues and the current trend according to their categories. In trend analysis, we have found that in former studies [1,2,11-13], researchers focused on generating complete code by sacrificing either in the 'naturalness' of natural language descriptions or the domain of generated source code. Following studies focused on resolving the restrictions by exploiting machine/deep learning techniques and statistical language models. Using these techniques, recent studies resolved both restrictions [8,10,15,17], but they sacrificed the completeness of the generated source code. However, these techniques show promising potential as we live in the era of flourishing data of source code and rooms for improvement lie in applying language models with source code data. One practical improvement we suggest is exploiting a deeply pre-trained representation of source code corpora and fine-tuning to different tasks. The research field of NLP showed that this approach has significantly improved many different tasks. By resolving these issues and keeping the freedom of domain and "naturalness," we believe this research field will bring a new paradigm to the software engineering discipline.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MIST) (No. 2018R1C1B6001919).

References

[1] D. Price, E. Riloffff, J. Zachary, and B. Harvey, "NaturalJava: a natural language interface for programming in Java," in *Proceedings of the 5th International Conference on Intelligent User Interfaces*, New Orleans, LA, 2000, pp. 207-211.

[2] D. Vadas and J. R. Curran, "Programming with unrestricted natural language," in *Proceedings of the Australasian Language Technology Workshop*, Sydney, Australia, 2005, pp. 191-199.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1-37, 2018.

[4] G. Neubig and M. Allamanis, "Modelling Natural Language, Programs, and their Intersection," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorial Abstracts*, New Orleans, LA, 2018, pp. 1-3.

[5] O. Pulido-Prieto and U. Juarez-Martinez, "A survey of naturalistic programming technologies," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1-35, 2017.

[6] G. L. White and M. P. Sivitanides, "A theory of the relationships between cognitive requirements of computer programming languages and programmers' cognitive characteristics," *Journal of Information Systems Education*, vol. 13, no. 1, pp. 59-66, 2002.

[7] C. Ghezzi and D. Mandrioli, "The challenges of software engineering education," in *Software Engineering Education in the Modern Age.* Heidelberg, Germany: Springer, 2005, pp. 115-127.

[8] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, Lille, France, 2015, pp. 2123-2132.

[9] P. Clark, B. Porter, and B. P. Works, "Km–the knowledge machine 2.0: user's manual," Department of Computer Science, University of Texas at Austin, Austin, TX, 2004.

[10] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, 2016, pp. 345-356.

[11] G. E. Heidorn, "An interactive simulation programming system which converses in English," in *Proceedings of the 6th Conference on Winter Simulation*, San Francisco, CA, 1973, pp. 781-794.

[12] G. Little and R. C. Miller, "Translating keyword commands into executable code," in *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, Montreux, Switzerland, 2006, pp. 135-144.

[13] R. Knoll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, 2006, pp. 542-559.

[14] A. Cozzie, M. Finnicum, and S. T. King, "Macho: Programming with Man Pages," in *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS)*, Napa, CA, 2011.

[15] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, 2018, pp. 933-944.

[16] W. Ling, E. Grefenstette, K. M. Hermann, T. Kocisky, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," 2016 [Online]. Available: https://arxiv.org/abs/1603.06744.

[17] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," 2017 [Online]. Available: https://arxiv.org/abs/1704.01696.

[18] T. J. Schriber, "Simulation using GPSS," University of Michigan, Ann Arbor, MI, 1974.

[19] S. Gulwani and M. Marron, "Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, Snowbird, UT, 2014, pp. 803-814.

[20] S. Shi, Y. Wang, C. Y. Lin, X. Liu, and Y. Rui, "Automatically solving number word problems by semantic parsing and reasoning," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, Lisbon, Portugal, 2015, pp. 1132-1142.

[21] S. Mandal and S. K. Naskar, "Natural language programing with automatic code generation towards solving addition-subtraction word problems," in *Proceedings of the 14th International Conference on Natural Language Processing (ICON)*, Kolkata, India, 2017, pp. 146-154.

[22] S. Chong and R. Pucella, "A framework for creating natural language user interfaces for action-based applications," 2004 [Online]. Available: https://arxiv.org/abs/cs/0412065.

[23] A. Begel and S. L. Graham, "Spoken programs," in *Proceedings of 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, Dallas, TX, 2005, pp. 99-106.

[24] M. Mefteh, A. Ben Hamadou, and R. Knoll, "Ara_Pegasus: a new framework for programming using the Arabic natural language," in *Proceedings of International Conference on Computing and Information Technology (ICCIT)*, Chittagong, Bangladesh, 2012, pp. 468-473.

[25] H. Lieberman and M. Ahmad, "Knowing what you're talking about: natural language programming of a multi-player online game," in *No Code Required*. Amsterdam, The Netherlands: Morgan Kaufmann, 2010, pp. 331-343.

[26] V. Le, S. Gulwani, and Z. Su, "Smartsynth: synthesizing smartphone automation scripts from natural language," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, Taipei, Taiwan, 2013, pp. 193-206.

[27] M. Landhaußer, S. Weigelt, and W. F. Tichy, "NLCI: a natural language command interpreter," *Automated Software Engineering*, vol. 24, no. 4, pp. 839-861, 2017.

[28] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Pittsburgh, PA, 2015, pp. 416-432.

[29] C. Quirk, R. Mooney, and M. Galley, "Language to code: learning semantic parsers for if-this-then-that recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Beijing, China, 2015, pp. 878-888.

[30] M. Raghothaman, Y. Wei, and Y. Hamadi, “Swim: synthesizing what I mean-code search and idiomatic snippet synthesis,” in *Proceedings of 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, Austin, TX, 2016, pp. 357-367.

[31] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, “Program synthesis from natural language using recurrent neural networks,” Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Rep. No. UW-CSE-17-03-01, 2017.

[32] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, “Nl2bash: a corpus and semantic parser for natural language interface to the Linux operating system,” 2018 [Online]. Available: https://arxiv.org/abs/1802.08979.

[33] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: generating structured queries from natural language using reinforcement learning,” 2017 [Online]. Available: https://arxiv.org/abs/1709.00103.

[34] R. Sirres, T. F. Bissyande, D. Kim, D. Lo, J. Klein, K. Kim, and Y. Le Traon, “Augmenting and structuring user queries to support efficient free-form code search,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2622-2654, 2018.

[35] V. Schlegel, B. Lang, S. Handschuh, and A. Freitas, “Vajra: step-by-step programming with natural language,” in *Proceedings of the 24th International Conference on Intelligent User Interfaces*, Marina del Ray, CA, 2019, pp. 30-39.

[36] H. Liu and H. Lieberman, “Metafor: visualizing stories as code,” in *Proceedings of the 10th International Conference on Intelligent User Interfaces*, San Diego, CA, 2005, pp. 305-307.

[37] P. Clark, P. Harrison, T. Jenkins, J. A. Thompson, and R. H. Wojcik, “Acquiring and using world knowledge using a restricted subset of English,” in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference*, Clearwater Beach, FL, 2005, pp. 506-511.

[38] P. Clark, W. R. Murray, P. Harrison, and J. Thompson, “Naturalness vs. predictability: a key debate in controlled languages,” in *Controlled Natural Language*. Heidelberg, Germany: Springer, 2009, pp. 65-81.

[39] K. Somasundaram and H. Swaminathan, “Automatic programming through natural language compiler,” in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, Las Vegas, NV, 2011.

[40] M. Soeken, R. Wille, and R. Drechsler, “Assisted behavior driven development using natural language processing,” in *Objects, Models, Components, Patterns*. Heidelberg, Germany: Springer, 2012, pp. 269-287.

[41] N. Kushman and R. Barzilay, “Using semantic unification to generate regular expressions from natural language,” in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Atlanta, GA, 2013, pp. 826-836.

[42] V. Harinarayan, A. Rajaraman, and A. Ranganathan, “Hybrid machine/human computing arrangement,” U.S. Patent 7197459, Mar 27, 2007.

[43] A. E. Cozzie and S. King, “Macho: Writing programs with natural language and examples,” 2012 [Online]. Available: https://www.ideals.illinois.edu/handle/2142/33791.

[44] M. Manshadi, D. Gildea, and J. Allen, “Integrating programming by example and natural language programming,” in *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, Bellevue, WA, 2013.

[45] T. M. Mitchell, “Generalization as search,” *Artificial Intelligence*, vol. 18, no. 2, pp. 203-226, 1982.

[46] J. Howard and S. Ruder, “Universal language model fine-tuning for text classification,” in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, Melbourne, Australia, 2018, pp. 328-339.

[47] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, K. (2018). “Bert: pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Minneapolis, MN, 2019, pp. 4171-4186.

[48] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, L. (2018). Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, New Orleans, LA, 2018, pp. 2227-2237.

Jiho Shin **https://orcid.org/0000-0001-8829-3773**

He received B.S. degree in computer science from Handong Global University in 2019. Since March 2019, he is with Information and Communication Engineering from Handong Global University as a M.S. student. His current research interests include defect prediction, automatic patch generation and NLP.

Jaechang Nam **https://orcid.org/0000-0003-1678-2185**

He received B.S. degree in computer science from Handong Global University in 2002, M.S. in computer science from Blekinge Institute of Technology in 2009, and Ph.D. in computer science and engineering from The Hong Kong University of Science and Technology in 2015. He is currently an assistant professor in the Department of Computer Science and Electrical Engineering, Handong Global University, Pohang, Korea. His research interests include mining software repository (MSR), software quality prediction, transfer leaning in software engineering.