Program Semantics-based Task Decomposition

Zhiming Wang
Software Engineering National University of Defense Technology Changsha, China
wangzhiming14@nudt.edu.cn

Abstract — Recently, automatic code generation technology has been extensively studied and gradually becomes an important part of software development. However, when facing coarse-grained, highly general task description, the current automatic code generation technology still cannot achieve good results. Therefore, we propose program semantics-based task decomposition technology. For a task description given by the user, natural language processing and deep learning technology are used to learn and understand it. Combined with the program semantics contained therein, we decompose the task into ordered subtask sequence. We evaluate our proposed model on constructed data set and achieve a BLEU-4 score of 18.13.

1. INTRODUCTION
With the rapid development of public code repositories on the Internet, such as GitHub and Stack Overflow, more and more high-quality code resources have become available, which provides new ideas for traditional software development, namely how to use existing massive code knowledge on the Internet, combined with increasingly mature data mining and machine learning technologies, to intelligently improve the efficiency and quality of the software development process, thus reducing the overhead of the software development process. Code generation is one of those important attempts.

Code generation technology refers to the automatic generation of corresponding program code according to the natural language description provided by the user or a specific protocol, which has far-reaching significance for the realization of software development automation. The current code generation techniques are mainly divided into two categories: Inductive Program Synthesis (IPS) and deep learning-based code generation. For a certain amount of <input, output> data pairs given by the user, the goal of IPS is to generate a code segment that can transform the given inputs into corresponding outputs. For example, in Microsoft Excel 2013 [1], its Flash Fill function applies IPS to automatically learn to write programs from a small number of examples provided by the user and achieves good performance. With the continuous development of deep learning technology, code generation methods based on deep learning have emerged in recent years. For example, DeepAPI [2] adapts a neural language model named RNN Encoder-Decoder to encode a word sequence (user query) into a fixed-length context vector, and generate an API sequence based on the context vector.

However, existing code generation techniques still have certain limitations. They can only understand the fine-grained task description (in natural language form) provided by the user, and generate a smaller-scale code fragment with single functionality. Therefore, when facing a coarse-grained, highly generalized task description, few good results can be achieved.

The aforementioned problem motivates us to design a task decomposition model that can transform a coarse-grained task description into several fine-grained task descriptions. For example, to implement
a certain task “Download a URL using GET”, we can actually decompose it into four subtasks as follows:

- create a new HTTP client
- set up the GET request
- execute the request in the client
- grab the response

Compared with the initial task description, the descriptions of subtasks are simpler, which can make the results of code generation more accurate.

To achieve program semantics based task decomposition, we first investigate the source code in popular open source communities such as Stack Overflow and GitHub, and find that external comment corresponding to the function can be seen as a task description of it, and the multiple ordered internal comments are the subtask descriptions. Therefore, we crawl the source code of the Java projects from GitHub, using natural language processing technology and program static analysis methods to extract the external comment and corresponding internal comments of functions, and converting them into the form \(<\text{task}, (\text{subtask}_1, \text{subtask}_2, ..., \text{subtask}_n)>\). Finally, we build a task decomposition model by using attention-based RNN Encoder-Decoder algorithm, and train it with constructed dataset. The trained model can predict the best subtask sequence for a given natural language task.

The main contributions of this paper are summarized as follows:
(a) We propose a program semantics-based task decomposition technology, which can transform a coarse-grained task description into fine-grained subtask description sequence, as a contribution to existing code generation methods.
(b) We crawl many high-star Java projects from GitHub and extract 30000 \(<\text{task}, (\text{subtask}_1, \text{subtask}_2, ..., \text{subtask}_n)>\) pairs from them. We evaluate our model on this dataset and achieve a BLEU-4 score of 18.13.

The remainder of this paper is organized as follows. We detail the data preparation in Section II. We introduce our proposed model in Section III. Finally, we represent our evaluation and conclude our work.

2. DATA PREPARATION
We crawl about 100GB of source code compression packages of more than ten-star Java projects from GitHub, and extract \(<\text{external comment}, \text{function code}>\) pairs from them. We learn from the external comments and internal segmentation comments of the code, and select appropriate data processing methods to obtain an effective data set to prepare for model training based on deep learning algorithms.

2.1. External Comment Processing
After extracting the external annotations, they need to be processed in two steps: first, preliminary screening, including deleting some unnecessary words and sentences, extracting key sentences, and replacing unique words, etc. The purpose is to obtain a pure natural language task description; Natural language processing includes operations such as word segmentation, converting English characters into lowercase, lemmatization, and generating syntax trees, and finally we extract the appropriate verb phrases in the annotations as functional descriptions.

2.1.1. Preliminary Screening
We first make sure that all the extracted functions are independent functions with external annotations, and then we exclude some classes and interface implementations by searching for characters such as "class", "interface", "enum", "abstract" in the function declaration, As well as iterations simply defining variables and abstract classes without specific function implementation. This step of screening ensures that the obtained functions are complete functions that only implement a single functionality.

According to the characteristics of external annotations, we take the following steps in the experiment for preliminary processing:
2.1.1.1. Delete all sentences that do not describe the function of the code snippet: The useless information is usually described by "@" plus a word, including "@throws", "@param", "@inheritDoc", "@see", "@category", "@return", etc. and their subsequent corresponding descriptions.

2.1.1.2. Delete all individual irrelevant characters: Exclude "*", "\t", "\n" and other non-alphanumeric characters, to get sentences that are nearly natural language descriptions.

2.1.1.3. Extract the first complete sentence in the comment as the target sentence: According to the Javadoc guide [3], programmers usually describe the function of subsequent code segments in the first sentence of the general comment.

2.1.1.4. Replace unique "words" in comments: These words include {code supertype}, {link AxisAlignedBB}, etc., which may have an impact on subsequent natural language processing.

2.1.2. Natural Language Processing
After completing the above-mentioned preliminary processing steps, a single sentence composed of pure natural language that can describe the functionality of a single function is obtained. By performing natural language processing on these sentences, more effective data expression forms can be obtained.

We first segment the sentence, and then convert all the characters into lowercase letters to unify some of the words, and then we perform lemmatization, which can restore any form of the vocabulary to a general form that can express complete semantics and not lose information. Finally, we perform part-of-speech tagging, and perform syntactic structure analysis by setting regular expressions, and finally extract verb phrases as functional descriptions.

2.2. Internal Segment Comments Processing
The internal comments are usually divided into multiple ordered pieces, which are mainly used by programmers to describe the functions of sub-code blocks in the code snippet. They can also be understood as sub-steps to complete the function, therefore they have very important reference value.

The processing of internal comments can be divided into three steps: First, the characteristics of internal comments in the source code are researched and analyzed, and screening rules are set to obtain a relatively clean internal comment data set; and then on the basis of the above data set internal comments are processed separately by natural language processing technology, including word segmentation, converting English characters to lowercase, using stop vocabulary to filter useless words, lemmatization, etc.; finally, the segmented comments within the same function are merged as a target sentence.

2.2.1. Filtering Rules Setting
By learning the characteristics of internal comments in a large amount of open source code, and setting up the following screening rules for various irregular comment writing:

2.2.1.1. Remove comments with source code.

2.2.1.2. Merge multiple lines that belong to a single comment.

2.2.1.3. Reserve only the outermost comments and remove other different levels of comments.

2.2.1.4. Reserve only English comments.

2.2.1.5. Remove comments with less than 3 words.
2.2.1.6. Set the minimum number of comments to 2.

2.2.1.7. Set the maximum number of comments to 10.
After filtering by the rules set above, the format of internal comments can be standardized.

2.2.2. Natural Language Processing
The processing of internal comments is different from external comments, mainly because the descriptions of internal comments are usually more colloquial, which makes it more difficult to extract verb phrases. We first perform word segmentation, then remove all words that begin with numbers, special characters, and punctuation. Then we convert all English characters into corresponding lowercase characters. We further carry out stop word screening. We use the stop word list in the work of Prefind [4], which included some nouns, verbs, modal verbs and prepositions. We also perform lemmatization. By filtering the useless words in the internal comments, the expression of a single comment is shortened without losing the key semantic information.

2.2.3. Comments Merging
After the natural language processing of all internal comments is completed, several segmented comments within the same function need to be merged and processed as a target sentence. In the experiment, multiple comments were merged in order by adding the character string ";;" after each comment. This method can make the data set have obvious segmentation characteristics, and the model can learn the characteristics better.

Finally, we construct a data set of size 15000 with high quality, which consists of pairs like <task, (subtask1, subtask2, ..., subtaskn)>.

3. THE PROPOSED MODEL
After multiple <task, (subtask1, subtask2, ..., subtaskn)> pairs are generated, we use RNN encoder-decoder model to learn relationships between task and subtasks. RNN encoder-decoder model include two sub-module: RNN encoder and RNN decoder. We use RNN encoder to encode task T into a vector c and use RNN decoder to decode vector c into subtasks.

3.1. RNN Encoder-Decoder
To start with, the RNN Encoder embeds task sentence T’s words into a distributed vector sequences [word1, word2, …, wordn]. Then, it learns the sequence into a fix-length vector c as follows. It sequentially inputs wordi in T and outputs a hidden state hi in formula of

\[ h_i = f(h_{i-1}, \text{word}_i) \]  \hspace{1cm} (1)

In formula (1), f denotes a non-linear function, h_{i-1} is the last output hidden state. And the final output h_n is the vector c which is the result of RNN Encoder.

After RNN Decoder obtain encoded vector c which have all information of task T, it generate each word wordi in subtask sequences [word1, word2, ..., wordn] in formula of

\[ h_i = f(h_{i-1}, c) \]  \hspace{1cm} (2)

\[ \text{word}_i = pr(h_i) \]  \hspace{1cm} (3)

In formula (2), f denotes a non-linear function, h_i is the i-th output hidden state which imply probability distributions of words, and in formula (3), pr is a prediction function which predict word according to the hidden state h_i.

To obtain multiple subtasks, we adopt beam search to generate most probable words according to hidden state h_i. Similarly, we update multiple hidden state h_i with corresponding last word.

3.2. Model Training
To make generated segment annotation sentence more similar to real annotation sentence, we train our model with a loss function in formula of
In formula (4), $N$ is size of training dataset, $T$ is length of the target sentence, and $cost_{it}$ is a cost function to count similarity between the $t$-th word in the $i$-th sentence and the corresponding real word of sentence. And $cost_{it}$ is computed with:

$$cost_{it} = - \log p_{\theta}(y_{it} | x_t)$$  \hspace{1cm} (5)$$

In formula (5), $\theta$ represents all parameters in RNN Encoder-Decoder model, and $P_{\theta}(y_{it} | x_t)$ imply the generation probability of the $t$-th generated word $y_{it}$ to the real sentence $x_t$.

### 3.3. Attention Mechanism

In process of decomposing task into multiple subtasks, each word in task sentence has different importance to generate words of subtasks. Thus, we use attention mechanism to measure such a importance. In concrete, for each output word, we will generate a unique vector $c_i$ instead of a shared vector $c$. $c_i$ is computed with:

$$c_i = \sum_{t=1}^{T} \alpha_{it} h_t$$  \hspace{1cm} (6)$$

In formula (6), $\alpha_{it}$ is a weight parameter imply the importance of $h_t$ to $c_i$, $h_t$ is the $i$-th output hidden state of RNN encoder.

### 3.4. Model Implementation

We implement our model based on OpenNMT framework which is an open source framework that integrates numerous deep learning models and algorithms. And the experiments were conducted on a computer with 1080 Ti GPU, running on Ubuntu 16.04. To train our model, we use stochastic gradient descent algorithm which can adjust learning rate according to learning process. Then we adjust model parameters to minimize loss function. After that, model obtain the best performance with model parameters as follows. The length of all word vector is set to 80, batch size is set to 100. And the dictionary size of task sentence and sub-task sentence are both set to 10000, such a dictionary keep the most frequent words and other words are set to "<UNK>".

### 4. EXPERIMENT AND EVALUATION

In the following experiment process, we use the constructed data set to train and test our proposed task decomposition model, and select the best model for evaluation. In the process of training and testing, the data set is randomly divided into three parts: training set, validation set and test set.

The training set contains 90% of the data randomly selected from the data set, which is used for model establishment and data fitting, and through continuous fitting to find the best model parameters. The validation set includes 5% of the data in the data set, and this 5% of the data does not overlap with the training set. The validation set verifies the effect of the model during the training process to prevent overfitting to the training set. The test set contains the remaining 5% of the data set. The test set is to evaluate the effect of the final model after training, so as to measure the performance of the model.

We use BLEU [5] to evaluate the accuracy of the subtasks generated by the task decomposition model. BLEU is an algorithm used to evaluate the quality of text translated from one natural language to another. BLEU is based on the n-gram model to calculate the similarity between two sentences. We use it to measure the similarity between a candidate sentence and the target sentence. The calculation method is shown as follows:

$$BLEU = BP \cdot \exp \left( \sum_{n=1}^{N} \omega_n \log p_n \right)$$  \hspace{1cm} (7)$$

In formula (7), $p_n$ is the proportion of the subsequence of length $n$ in the candidate sentence in the target sentence. $N$ is the parameter $n$ in n-gram, which is set to be 4 in our work. $w_n$ is the weight of $p_n$. 
BP is a penalty factor set for candidate sentences whose length is shorter than the target sentence. The higher the value of BLEU, the more similar between the two sentences.

Finally, our model achieves a BLEU-4 score of 18.13. We also make a statistics of the data and we find that the average length of the task is 12.49, the average length of the subtask is 8.31, each task average corresponds to 2.78 subtasks and the total number of the data set is 15000.

To better present our results, we display several examples in the following table.

**TABLE I. GENERATED SUBTASK SEQUENCE EXAMPLES**

| Task                                      | Generated Subtask Sequence                                      |
|-------------------------------------------|-----------------------------------------------------------------|
| create a thread safe client               | set up the http part of the service ;; register the http protocol scheme ;; set some client http client parameter defaults ;; |
| create the GUI and show it                | create and set up the window ;; create and set up the content pane ;; display the window ;; |
| resample the captured photo to fit the screen for better memory usage. | get the device screen size information ;; get the dimensions of the original bitmap ;; determine how much to scale down the image ;; decode the image file into a bitmap sized to fill the View ;; |
| delete the image file for a given path    | get the file ;; delete the image ;;                            |
| update the blocks bounds based on its current state | prepare the variables ;; prepare block meta and normalize meta ;; set block bounds depending on parameters ;; |

As shown in the table, in many cases, we can easily understand the subtask sequence generated by the model and they are very close to ground truth, which indicates that our model can successfully decompose several tasks.

5. CONCLUSION AND FUTURE WORK

In this paper, we propose program semantics-based task decomposition technology to help existing automatic code generation. For a task description given by the user, natural language processing and deep learning technology are applied to learn and understand it. We take corresponding program semantics into consideration and construct a data set based on external and internal comments. We propose an attentional RNN Encoder-Decoder model and finally the model can decompose the given task into ordered subtask sequence. We make a evaluation of our proposed model and find that it achieves a BLEU-4 score of 18.13.

We find that the proportion of internal comments in the source code is very small, which may affect the effect of the model. We would like to set some rules to automatically segment the source code, and use code summarization to generate comments for each segment. In this way, we can obtain a larger data set and achieve a better result.

REFERENCES

[1] Gulwani S, Esparza J, Grumberg O. Programming by Examples (and its applications in Data Wrangling) [C]. Verification and Synthesis of Correct and Secure Systems. IOS Press, 2016, in press.
[2] Gu X, Zhang H, Zhang D. Deep API learning[C]. Proceedings of the 24th ACM SIGSOFT International Symp. on Foundations of Software Engineering. ACM, 2016: 631-642, in press.
[3] Javadocguidance[EB/OL]. http://www.oracle.com/technetwork/articles/java/index137868.html
[4] Sridhara G, Hill E, Muppaneni D, et al. Towards automatically generating summary comments for Java methods[C]// ASE 2010, Ieee/acm International Conference on Automated Software Engineering, Antwerp, Belgium, September. DBLP, 2010:43-52, in press.
[5] Papineni K, Roukos S, Ward T, et al. Bleu: a Method for Automatic Evaluation of Machine Translation[C]. meeting of the association for computational linguistics, 2002: 311-318, in press.