Breaking Down High-Level Robot Path-Finding Abstractions in Natural Language Programming

Yue Zhan(✉) and Michael S. Hsiao

Bradley Department of Electrical and Computer Engineering, Virginia Tech,
Blacksburg, VA 24060, USA
{zyue91,hsiao}@vt.edu

Abstract. Natural language programming (NLPr) allows people to program in natural language (NL) for specific domains. It poses great potential since it gives non-experts the ability to develop projects without exhaustive training. However, complex descriptions can sometimes have multiple interpretations, making program synthesis difficult. Thus, if the high-level abstractions can be broken down into a sequence of precise low-level steps, existing natural language processing (NLP) and NLPr techniques could be adaptable to handle the tasks. In this paper, we present an algorithm for converting high-level task descriptions into low-level specifications by parsing the sentences into sentence frames and using generated low-level NL instructions to generate executable programs for pathfinding tasks in a LEGO Mindstorms EV3 robot. Our analysis shows that breaking down the high-level pathfinding abstractions into a sequence of low-level NL instructions is effective for the majority of collected sentences, and the generated NL texts are detailed, readable, and can easily be processed by the existing NLPr system.

Keywords: Natural language processing · Natural language programming · Program synthesis · LEGO Mindstorms EV3

1 Introduction

The field of robotics has made significant strides because of the growth of market demands in recent years. However, despite the growing interest in educational robots, the time-consuming learning process and the steep learning curve of programming robots still challenge young robotics enthusiasts. Natural language programming (NLPr) offers a potential way to lower the bar of entry by allowing the users to "program" the robot using natural language (NL). The readability and expressive nature of natural language make it an ideal way to simplify the learning process. Though promising for this use case, NLPr has several challenges of its own. First, NL texts used to give instructions are typically low-level (LL) specifications to ensure precision and completeness. For example, the movement specifications used in the NLPr system for LEGO Mindstorms EV3 robot in

© Springer Nature Switzerland AG 2021
M. Baldoni and S. Bandini (Eds.): AIxIA 2020, LNAI 12414, pp. 280–297, 2021.
https://doi.org/10.1007/978-3-030-77091-4_18

the work [24] are categorized as a controlled natural language (CNL) [7]. The movement sentences used in the system are object-oriented sentences like "*The robot goes forward/backward/left/right ...*". The requirement to use such low-level specifications makes the process of directing the robot more difficult for novice users, as they would rather give a high-level instruction such as *"The robot moves from point A to point B"* than to list out every individual step the robot must take. Unconstrained NL texts are highly flexible and expressive but can sometimes be ambiguous. Designing a language model for NLPr to cover all of the language structures in NL is extremely difficult, if not impossible [2]. As such, it would be a huge benefit for NLPr tasks if the information in a higher-level abstraction can be effectively extracted and used to generate a sequence of precise, unambiguous lower-level sentences that explain the intention and plans the proper actions. Suppose the information related to the robot tasks can be extracted. In that case, the language structures that need to be covered in the domain-specific function library and lexicon can be simplified, and the existing NLPr system can be directly adapted with fewer necessary modifications to handle the high-level (HL) NL abstractions, as shown in Fig. 1.

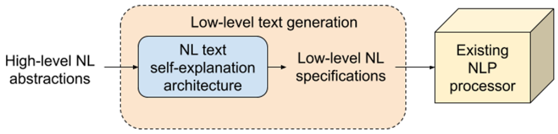

Low-level text generation
High-level NL abstractions
NL text self-explanation architecture
Low-level NL specifications
Existing NLP processor

Fig. 1. High-level NL to low-level NL transformation

The key challenge addressed in this paper is effectively extracting semantic information from high-level sentences to synthesize low-level pathfinding NL instructions. In order to demonstrate our proposed low-level text generation process, we use a robot pathfinding task, in which a robot must find an optimal path between two points while avoiding obstacles along the way. To succeed at this task, our system must generate a sequence of low-level instructions that take the robot to its goal while minimizing the time cost and the number of actions taken by the robot. Once our system identifies a path based on the high-level input, it outputs a sequence of low-level NL to an existing NLPr system [24], which then generates the executable program for the LEGO Mindstorms EV3 robot.

2 Previous Work

Due to its promise of better ease of use and improved human-computer interaction, the foundations of NL based programming for robotics have been well established. In [8,13], an NLPr system that navigates a vision-based robot with an instruction-based learning method is presented. In these systems, robot-understandable procedures are generated from command-like NL instructions based on a set of pre-programmed sensory-motor action primitives and routes in a miniature city map. Users can give instructions based on available primitives to the robot when facing an unknown route with a human-robot interaction interface. In the work [23], a Vision-language robot navigation (VLN) system that combines the vision information and descriptive NL commands reasoning using a data-based model is proposed for in-home environments. When the NL instructions are given, a sequence of actions is generated by the reasoning navigator. Gathering sufficient data on various environments for model training purposes could be costly. In the work [15], a probabilistic combinatory categorial grammar (PCCG) based parser is used to translate procedural NL commands into logic-based Robot Control Language (RCL) for robot navigation. In [24], a grammar-based Object-Oriented Programming Controlled Natural Language (OOP-CNL) model is used to translate NL sentences into executable code for LEGO robots. In this work, the NLPr program synthesis system utilizes contextual and grammatical information to derive desired robot functionalities with a domain-specific function library and lexicon. While the language model used here can process more complex sentence structures, such as conditional statements, the sentences used for navigating the robot are still at a lower level.

There has been a significant amount of work done in the field of NLPr program synthesis, and most of this work has been focused on solving domain-specific problems. The work in [3] emphasizes the importance of NLP techniques in analyzing textual contents in software programs. The authors propose a system called Toradocu, which they developed using Stanford parser and a pattern and lexical similarity matching that coverts Javadoc comments into assertions, and a system called Tellina, which is trained with an RNN [10] to generate bash commands and use these systems to illustrate the potential of program synthesis with NL texts. The Metafor platform [11,18] is a descriptive NLPr system that can convert NL components into class descriptions with associated objects and methods. This work takes advantage of the NL parsing toolkit MontyLingua, mixed-initiative dialog, and programming by example techniques. The authors state that modern parsing techniques and the integration of common sense knowledge can help developers link humans' narrative capacities with traditional programming languages. However, the programs generated by Metafor are not directly executable. Another work, DeepCoder [1] extends the programming by example framework *Learning Inductive Program Synthesis* (LIPS)[17] into a big data problem. DeepCoder generates a sequence of SQL-like function calls for given integer input-output examples by training a neural network to predict possible mathematical properties. However, the generated function calls are basic and low-level. In work [5], an NLPr video game design system translates object-oriented English sentences into JavaScript game code. A hybrid context and grammar analysis is used. Conditional statements also can be handled in this system.

Text generation is a topic of interest in NLP research, and it is also receiving attention in the domain of robotics. A number of systems have worked towards explaining robot behavior, including verbalizing the robot's navigation decisions

[19,21] and explaining robot policies by generating behavioral explanations in NL [4]. The idea of generating low-level robot NL specifications based on robot paths presented in this paper is similar to these works: breaking down abstracted robot missions into sequential steps describing robot behaviors. However, instead of being used to explain the navigation to humans, the generated low-level NL texts are used for NLPr program synthesis.

3 Problem Formulation and System Design

3.1 High-Level to Low-Level (HL2LL) System Overview

Parsing and understanding the semantic meanings of high-level abstractions have been a significant challenge in NLP and NLPr research due to their complex linguistic nature. Just like explaining a complex concept to a child, one needs to break the concept down to a sequence of discrete, straightforward, and actionable steps for machines to understand. In this work, particularly, the HL2LL mechanism is built upon a domain-specific library; in this case, the LEGO robot functions. The OOP-CNL language model L [24] is used to extract the function information from NL inputs and to match a suitable combination of robot functions in this work. In a nutshell, when the function information extracted from the high-level abstraction contains motion language features that cannot be translated into individual functions in the function library $\mathcal{F}$, the system would further search for identifying high-level abstractions, like color line tracking or moving to specific mission regions. The high-level abstractions can be explained using a set of low-level specifications. For example, *"The robot moves forward 10 in."* is an example of a low-level specification, while *"The robot walks to M4 from M1."* is considered a high-level abstraction since it can be described using a set of low-level instructions. The transformation process, shown in Fig. 2, consists of four steps:

1. Parse the high-level abstraction: Identify the task details from given input sentences.
2. High-level abstraction to path: Find a qualified path from the *source* to the *target* based on the given high-level abstraction using the algorithm in Sect. 3.3.
3. Path to low-level NL specifications: Generate a set of low-level NL specifications that describe the actions needed for the robot to follow the qualified path.
4. Low-level NL specifications to code: Translate low-level NL specifications into executable codes using the NLPr system.

3.2 Map Representation

We model our robot's task after the First LEGO League (FLL)[1] competition, with an 88″ × 44″ *Mission Map* based on the FLL 2018/2019 official competition

[1] https://www.firstlegoleague.org/.

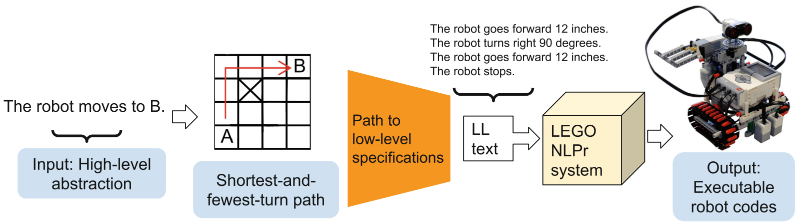
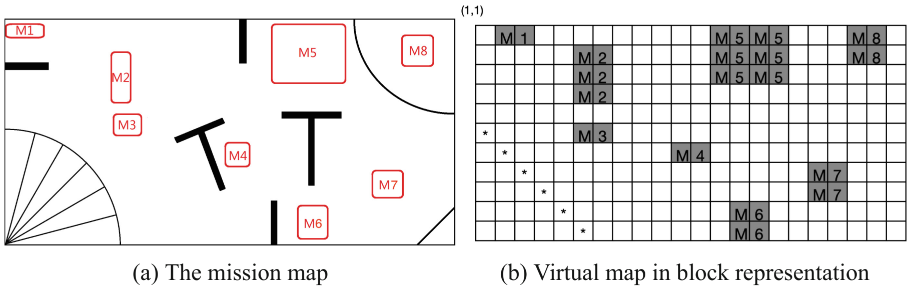

The robot moves to B.
Input: High-level abstraction
A
B
Shortest-and-fewest-turn path
Path to low-level specifications
The robot goes forward 12 inches.
The robot turns right 90 degrees.
The robot goes forward 12 inches.
The robot stops.
LL text
LEGO NLPr system
Output: Executable robot codes

Fig. 2. System overview

arena serving as our robot's environment, shown in Fig. 3a. The arena contains eight mission regions, denoted using red blocks and several thick black lines on the map, which can be recognized using the robot's color sensor. The *Base* located at the bottom left is the required starting point for each run. In this paper, we focus on the task of planning a path for the robot between specified mission regions. Some other actions involving motor and sensor usages can be performed in addition to navigation, as described in the LEGO NLPr system [24].



(a) The mission map (b) Virtual map in block representation

Fig. 3. Virtual game maps

In order to simplify pathfinding, we break the *Mission Map* into grid squares, as shown in Fig. 3b. We denote this grid representation the *Virtual Map*. In the *Virtual Map* asterisks denote the edge of the start region and mission regions are represented by mission blocks. Mission blocks are shaded cells of the form M_n where the n represents the mission index. Each grid block corresponds to a block with size of $4'' \times 4''$. The top left corner of the map is initialized with coordinate $(1, 1)$. If the mission regions are treated as block-like obstacles, and the robot is restricted to movement in the cardinal directions, the task of pathfinding in the *Virtual Map* can be treated as a 2D Manhattan pathfinding problem.

3.3 Lee's Algorithm and Its Adaption

Lee's Algorithm [9] is one of the most effective breadth-first search (BFS) based single-layer routing methods for finding the shortest paths in a Manhattan graph.

Lee's Algorithm searches for the target from the source using a wave-like propagation. With a source block S and a target set of adjacent blocks T, there are two main phases in Lee's Algorithm:

1. **Search**: Begin by labelling block S as k, where $k = 0$. Fill the valid neighbors of blocks labeled k (not already filled and not outside of the map) with label $k + 1$. Proceed to step $k + 1$, repeating the previous process until either the destination T is reached or there are no more valid neighbors.
2. **Retrace**: Once T has been reached, trace backward to build the path from T to S by following the descend of k from k to 0. It is possible that multiple equal-length paths exist between S and T.

Lee's Algorithm can be modified to break ties between equal-length paths in favor of the path with the fewest turns, as shown in Algorithm 1 [16]. By minimizing the number of turns that the robot makes, we reduce the number of NL sentences our system must generate and the accumulation of navigation errors that occur as the robot turns. In the **Search** process, the direction and coordinates are recorded for the **Retrace** phase's reference. An alternative method approach would be to rank paths first by the number of turns taken and only then consider the overall path length, effectively trading off reduced turning time for potentially longer paths [25]. However, FLL players need to finish as many tasks as possible within a given time limit, and as such, we prefer to rank by path length first. Figure 4 shows an example of a grid's state after Algorithm 1 has been executed. Although there are multiple equal-length paths in the grid, the path highlighted in green is chosen by the adapted Lee's Algorithm because it has the fewest turns among the eligible shortest paths.

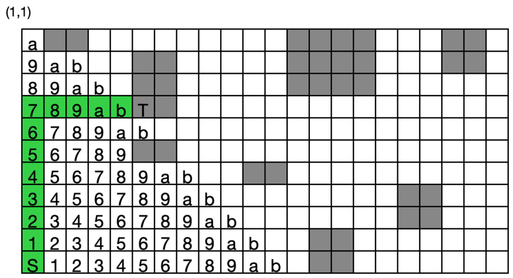


Fig. 4. Finding a path from the Base to M_2

3.4 Path Information Extraction for NLPr

Information extraction (IE) [6] in NLP is the process of converting raw text into a form that can be easily processed by machines. A task-driven domain-specific function library $\mathcal{F}$ is used to narrow down the space of function matching for program synthesis in this study. The function library $\mathcal{F}$ includes actions that a LEGO robot can perform with the supported sensors and motors. The key to

Algorithm 1 Shortest-and-fewest-turn path: Search and Retrace

```
 1: procedure SEARCH($current\_point$, $target\_point$)
 2:     $queue.push([source\_point, 0])$
 3:     while $queue$ do
 4:         $current\_point, counter \leftarrow queue.popleft()$
 5:         $i, j \leftarrow current\_point.i, current\_point.j$
 6:         $emap[i][j] \leftarrow counter$                          ▷ label the current point
 7:         $queue.push(neighborsOf(i, j), counter + 1)$ if valid
 8:         if any neighbor reaches the $target\_point$ then
 9:             $goal\_point \leftarrow (i, j)$, break                          ▷ path found
10:     if $goal\_point = source\_point$ then
11:         return                                                  ▷ no such path exists
12:     else
13:         save $current\_dir$
 1: procedure RETRACE($current\_point$, $source\_point$)
 2:     $get\_dir \leftarrow current\_dir$
 3:     while $current\_point \neq source\_point$ do
 4:         $i, j, id \leftarrow current\_point.i, current\_point.j, emap[i][j]$
 5:         $L\_id, R\_id, U\_id, D\_id \leftarrow neighbors(emap[i][j])$ if exists
 6:         if $get\_dir \in [L, R, U, D]$ and $(X\_id = id - 1)$ then       ▷ X: dir as id↓ along
    $get\_dir$
 7:             update $i, j$
 8:         else
 9:             compare to neighbors in different dirs
10:             update $i, j, get\_dir$
11:         $current\_point \leftarrow (i, j)$
12:         $path.push(current\_point)$
```

parsing a sentence's semantic meaning is to split the sentence into sentence frame components and identify the dependency relations in and between each frame. A grammar-based OOP-CNL model $\mathcal{L}$ [24] is used to construct an intermediate representation for pathfinding based on part-of-speech (POS) tags [22] and parse information using NLTK toolkits [12], defined as:

$$\mathcal{L} = (O, A, P, R) \tag{1}$$

where O stands for the objects in the arena, A represents the robot actions, P indicates the adjectives or adverbs affiliated with the objects and actions, and R represents the requirements or conditions for the objects or the actions.

In order to provide sufficient information for program synthesis for the task-driven robot NLPr system, the following sentence components must first be identified: the object O, the action A, their corresponding properties P, and the conditional rules R, if any exist. After an initial preprocessing step based on lemmatization and tokenization, keywords from the lexicon, such as sensor names, sensor and motor port numbers, and mission region names are identified. Then the sentence tokens are categorized based on grammatical tags and dependency relations. For example, in the input sentence "*A happy robot goes* *to M2.*" O is the `robot`; A is `go to M2`; P is a Boolean state `happy`, and the R is that the expression `happy==True` must evaluate to true in order for the object to perform the action, as shown in Fig. 5. In an ideal world, the combination of $OAPR$ extracted from a sentence would correspond to exactly one function in the function library $\mathcal{F}$. However, due to the ambiguous nature of NL, there exist sentences for which $OAPR$ either cannot be mapped to any function in the library and there exist sentences for which $OAPR$ can map to multiple functions. These sentences pose a problem because if they are passed to the downstream NLPr system, the system could generate a program that does not perform the action the user intended. One way to prevent passing these sentences downstream is to use a formal validation step, which can provide early detection of such ambiguous sentences. The validation of input sentences is done with a formal analysis engine powered by a context-sensitive hierarchical finite-state machine (HFSM), which will be introduced in Sect. 3.7.

In a robot path finding task, when mission regions are detected in the sentence, an error-checking step is invoked to detect any underlying errors in the text, as described in Algorithm 2. The object and action pairs identified in this step continue to a function matching process in the function library $\mathcal{F}$. The object `robot` and action `go` match the pathfinding function `find_path(start,target)` instead of the function `move(dir,num,unit)` because of the presence of the target `M2` in this example.

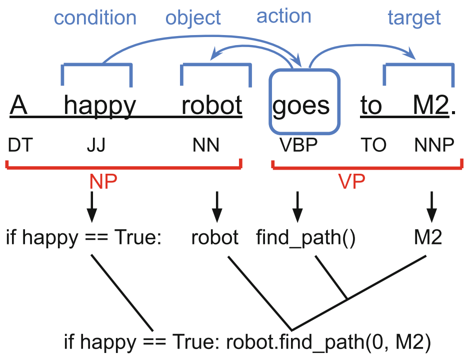


Fig. 5. Parsing a sentence and constructing the intermediate representation.

For our robot application, the number of object and action combinations is finite. For sentences with no ambiguity or errors, each $\mathcal{L}$ should have only one valid match in the finite function library $\mathcal{F}$. If the system fails to identify such a 1:1 match in the function library, the system will generate an error message with diagnostic information to help users to debug their input. When multiple objects, actions, or interpretations exist, the pre-defined higher priority functions will be chosen to ensure a sample program can be generated. For example, the sentence "*The robot goes straight to M3.*" maps to function `move(forward,0,0)` and function `find_path(0,M3)`. As such, the system cannot determine the user's intention. The system responds to this situation by generating a warning message, notifying users that "straight" is ignored for this conflict. Rather than not produce any low-level output at all, the system produces an output based on the `find_path` function, as it is a higher priority action.

When multiple mission regions are present, the pathfinding process needs to be split into steps. Each input sentence describing robot navigation may contain one midpoint and one avoid-point. For example, in the sentence "*If the robot sees an obstacle in 20 in., it goes to M7 through M3 but avoids M4*", the path is parsed into two steps with the midpoint (through (`M3`)), the target (to (`M7`)), and the avoid-point (avoids (`M4`)), under the condition (`if ultrasound_sensor()<20 inches`).

Algorithm 2 Check for errors in the pathfinding sentence

```
 1: procedure CHECK_ERRORS($tokens$)
 2:     $tokens, unknowns = tokens.validate(lexicon)$
 3:     if $unknowns$ then
 4:         Warning: Skipping detected unknown tokens.
 5:     $obj, act \leftarrow tokens.intersection(obj\_dict, act\_dict)$
 6:     if $obj \neq robot$ or $act \neq find\_path$ then                  ▷ mismatch
 7:         Error: not valid combination
 8:     $missions \leftarrow tokens.intersection(emap)$        ▷ get all mission regions in the sentence
 9:     if $len(missions) \geq 4$ then
10:         Error: too many mission regions in one sentence. Consider re-write.
11:     $source, target, mid\_point \leftarrow dependency(tokens)$
12:     if $!target$ or any $mission \in missions$ unsigned then
13:         Error: no valid target or dangling tokens
14:     else
15:         return [$robot.find\_path(source, mid\_point, target)$]
```

Multi-conditional statements can be handled in such a language model $\mathcal{L}$ by processing each condition as a Boolean statement and each action separately. For the example shown in Fig. 6, the sentence is processed into an *if* statement with 2 conditions: condition 1 (NP (`color sensor`) VP (`see black`)), condition 2 (NP (`robot`) VP (is `happy`)), and action (NP (it) VP (`move to M2`)). The reference relation between `it` and `robot` is done by contextual analysis on current and previous contents combined with function library restrictions. e.g. `robot` is chosen because of it matches with the action `move` behavior both contextually and functionally.

While an action might have several interpretations, the functions implied in a sentence are limited by the task-driven domain-specific function library. For example, in the sentence "*When the robot sees red at M1, it will speed up and go through M2 to reach M3.*", the `color` subject indicates a color sensor is needed.

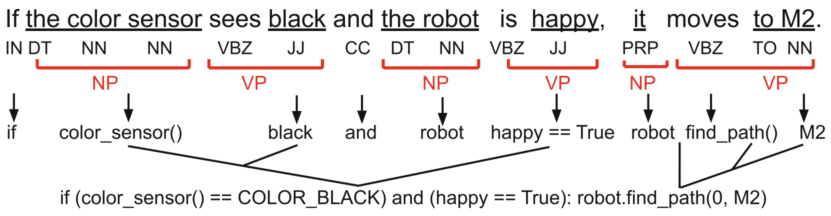

If the color sensor sees black and the robot is happy, it moves to M2.
IN DT NN NN VBZ JJ CC DT NN VBZ JJ PRP VBZ TO NN
NP VP NP VP NP VP
if color_sensor() black and robot happy == True robot find_path() M2
if (color_sensor() == COLOR_BLACK) and (happy == True): robot.find_path(0, M2)

Fig. 6. Complex sentence example

Similarly, "see a wall" indicates the ultrasound sensor usage and "touch a wall" indicates the touch sensor usage.

3.5 Path to Low-Level Sentence

Once a path between S and T is identified, a sequence of low-level NL sentences describing the corresponding step-by-step actions needed to navigate the LEGO robot is generated. A grammar-based formalization method is used to construct the object-oriented low-level NL sentences. The generated NL texts will be fed to an NLPr system for further translation, as opposed to being intended for humans to read. Our proposed method does not require a large dataset for training and can be adapted to other high-level abstractions when a suitable function library is available.

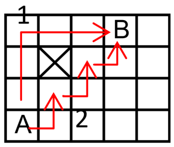


Fig. 7. The robot moves from A to B.

If the robot starts off facing North/up, path 1 in Fig. 7 is described in low-level NL specifications as:

Path 1: $[4, 1] \rightarrow \ldots \rightarrow [1, 1] \rightarrow \ldots \rightarrow [1, 4] \Rightarrow$ The robot goes forward 12 inches. The robot turns right 90 degrees. The robot goes forward 12 inches.

The pseudocode in Algorithm 3 illustrates the above path-to-sentence conversion. First, every two neighboring coordinates in the path array are compared to detect turns and step numbers in each turn. The function `compare((pre_row, pre_col), (row, col))` returns `state` that determines if the robot needs to turn. If not, it means the robot still follows the previous direction `pre_state`. The counter records the number of steps in the current direction. Once a turn occurs, a set of NL sentences is generated based on the number of steps, recorded direction, and previous state, i.e., the function `path2NL(pre_state, dir, count)` generates the NL sentences for each turn. We then update the direction and reset the counter for the next steps.

Algorithm 3 Path to NL Generation

```
1: procedure NL Text Generation($path$, $direct$)
2:     $total\_step, (pre\_row, pre\_col) \leftarrow len(path), path[0]$        ▷ total number of steps
3:     $counter, state, pre\_state \leftarrow 0, 0, 0$
4:     for $i$ in range($1$, $total\_step$ ) do
5:         $row, col \leftarrow path[i]$
6:         $state \leftarrow$compare($(pre\_row, pre\_col), (row, col)$)
7:         if $state = pre\_state$ then
8:             $counter$ += 1
9:         if $state \neq pre\_state$ or $i = total\_step - 1$ then
10:            $NL\_text \leftarrow$path2NL($pre\_state, dir, counter$)
11:            update($dir$), $counter \leftarrow 1$
12:            NL2Code($NL\_text$)        ▷ NL texts to code
13:        $pre\_state, pre\_row, pre\_col \leftarrow state, row, col$
```

3.6 Generating Code from NL Specifications Using the NLPr System

The LEGO NLPr program synthesis system [24] generates executable text-based programs directly from the NL input instead of the graph-based programs typical of LEGO robots. The input English Code (EC) is processed into intermediate representations using NLP techniques like, lemmatization, tokenization, categorization, and a function matching procedure. Such intermediate representations contain information extracted from the input that indicate the desired functions that need to be translated into formal program snippets. These intermediate representations are used for program synthesis and producing feedback or error information for users. The NL-to-code program synthesis system, `NL2Code(NL_text)` in Algorithm 3, calls the functions that handle the conversion of generated low-level NL specifications into executable programs. A set of robot motion functions in $\mathcal{F}$ are combined to synthesize the output program based on the intermediate representations. For example, the sentence "*The robot goes forward for 12 in.*" can be represented by `robot.move(forward,12,inch)`. This representation is translated into 28 lines of code.

3.7 Formal Validation Using HFSM

Finite-state machines (FSMs) are a powerful formal validation technique widely used in NLP applications such as IE and natural language parsing [14]. An FSM is an automaton with a finite number of transition states and terminal states. The transitions from one state to another are triggered with a predetermined

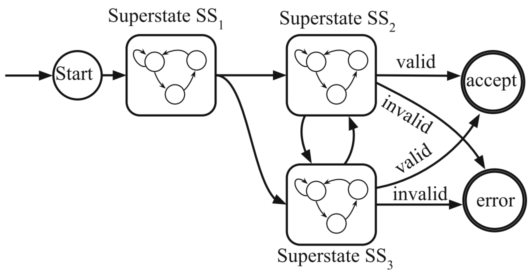


Fig. 8. Example HFSM

set of coded instructions [20]. At each state, the relevant FSM path to the next state is determined by the next input token in the sequence. Accurate translations are critical for NL-based robot program synthesis as misunderstanding the input's intention might lead to physical damage to the robot. The deterministic properties of FSMs help generate more trustworthy intermediate semantic representations and also help detect errors in the input, both of which contribute to less error-prone results for the NLPr system. In our LEGO NLPr system, the validation process in Algorithm 2, is powered by a *context-sensitive* hierarchical FSM based formal validation engine. This formal validation engine helps us to both validate input sentences and generate error messages whenever an error state is reach.

The FSM's hierarchical structure reduces the complexity of the system and allows us to specify the system more in detail by breaking the state machine into several *superstates*, denoted as SS_i, where $SS_i \in \mathbb{SS}, 1 \leq i \leq m$, as shown in Fig. 8. A *superstate* represents a cluster of one or more substates, as shown in Fig. 8. As mentioned above, the LEGO NLPr system is capable of handling conditional statements. To avoid mistranslation, the parsing process is split into two separate HFSMs with one for the condition and one for the action in a sentence. Take Boolean variable checking as an example: the conditional statement `if` "the robot is happy" refers to an expression that checks to see if the variable *happy* is `True`, while the action statement "the robot is happy" refers to a variable assignment that assigns *True* to `happy`.

For the robot pathfinding task we focused on in this paper, the transition from the robot *superstate* SS_1 to the pathfinding *superstate* SS_5 is triggered when a valid mission region name M_x is identified, as shown in Fig. 9. Within the pathfinding *superstate* SS_5, the target, source, midpoint, and avoid-point are identified. The formal validation within the *superstate* will check if any errors exist, such as illegal mission region names that are not registered on the mission map or having two target regions.

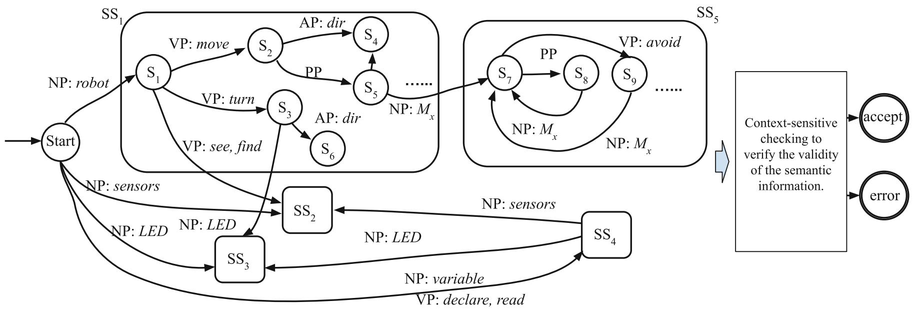


Fig. 9. Simplified robot action HFSM with some states omitted.

In order to guarantee the validity of the information extracted by the HFSM, we include a context-sensitive semantic checking based case analysis prior to transitioning to any terminal state, a step not present in conventional FSMs. For example, the case analysis would report errors and leads to the *error* terminal state if there are any conflicts between the target, source, midpoint, and avoid-point.

4 Experimental Results

We evaluate our system's performance on a set of 162 robot pathfinding related descriptions. These descriptions were collected manually by the authors, and they collectively describe movements between all eight mission regions. Each description consists of one or more sentences.

Our system successfully translates all 36 descriptions with 2 or fewer mission regions, resulting in programs that navigate the robot on the shortest path with the fewest turns between the source and the target. Of the 56 descriptions that navigate between three mission regions, 91.1% of the generated programs are correct. Our system performs worse on descriptions with more complicated structures, namely those involving more than three mission regions, with only 68.6% of the 70 such descriptions being translated into programs that conform to the original semantic meaning of the descriptions. Overall, our system correctly translated 135 (83.3%) of the 162 collected descriptions. Some example descriptions and their corresponding number of lines of code generated are shown in Table 1.

These results show that our proposed high-level abstractions to low-level NL instructions transformation system can successfully translate the large majority of the collected high-level robot navigation task sentences into low-level instructions for producing executable programs. This supports our hypothesis that with the POS tagging and a domain-specific function library and lexicon, the objects, actions, and targets in $\mathcal{L}$ can be effectively identified and useful intermediate representations for further program synthesis can then be generated.

However, despite our system's strengths, it still struggles with more complicated sentence structures due to the ambiguous and expressive nature of NL. One such description that poses a challenge for our system is "*The robot wanders through M1 M2 and M3.*" This description cannot be translated properly because there is no clear indication of the robot's source and target. As this description would be difficult for a human to convert to low-level instructions, it is understandable that the system fails to translate it correctly.

High-level robot navigation abstractions are translated into varying numbers of lines of code depending on the complexity of the NL instructions. When an NL description includes information that the system cannot handle, a best-guess program skeleton and accompanying debugging feedback are generated.

Table 1. English code examples with corresponding number of lines of code generated

Eg#	English Code Examples	# of lines
1	The robot goes from M5 to M3	99
2	The robot starts with facing to the right The robot goes to M8 from M1 but avoids M2	109
3	The robot goes to M1 without going through M2 via M3	81
4	If the robot is happy, it goes to M2	67
5	When robot does not see the red line, it goes straight to M3 Otherwise, it follows the red line	105
6	If the robot sees an obstacle in 20 in., it goes to M7 through M3 but avoids M4	108
7	If the ultrasound sensor sees a ball in 5 in., the robot is happy A happy robot goes to M7 though M3 but avoids M4	112

4.1 Case Study

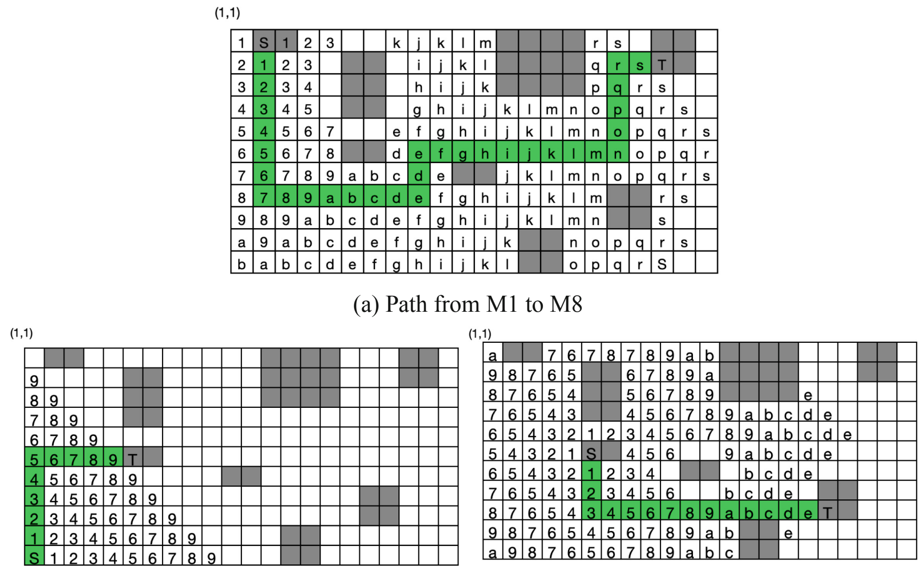
Example 1. 109 lines of code are generated for Example description #2 in Table 1 for navigating from `M5` to `M3`, shown in Fig. 10a.

Generated path $[1,2] \rightarrow \cdots \rightarrow [7,2] \rightarrow \cdots \rightarrow [7,8] \rightarrow [6,8] \rightarrow \cdots \rightarrow [6,17] \rightarrow [5,17] \cdots \rightarrow [2,17] \cdots \rightarrow [2,19]$

Generated low-level instructions "The robot turns right 90 degrees. The robot goes forward 24 in. The robot stops. The robot turns left 90 degrees. The robot goes forward 24 in. The robot stops. The robot turns left 90 degrees. The robot goes forward 4 in. The robot stops. The robot turns right 90 degrees. The robot goes forward 36 in. The robot stops. The robot turns left 90 degrees. The robot goes forward 16 in. The robot stops. The robot turns right 90 degrees. The robot goes forward 8 in. The robot stops."

Example 2. Paths for sample #7 in Table 1 are shown in Figs. 10b and 10c. Sample #7 in Table 1 is a multiple-phase pathfinding task, and as such our system must compute two paths. Note that when the robot reaches mission region #3, the robot is pointing to the East/right. Therefore, the second path starts with turning to the right only 90 degrees rather than turning 180 degrees. The second sentence's robot movements would only be performed when the state `happy` is true from the last sentence.

Generated path 1 $[11, 1] \rightarrow \cdots \rightarrow [6, 1] \rightarrow \cdots \rightarrow [6, 6]$
Generated path 2 $[6, 6] \rightarrow \cdots \rightarrow [9, 6] \rightarrow \cdots \rightarrow [9, 17]$
Generated low-level instructions "The robot goes forward 20 in. The robot stops. The robot turns right 90 degrees. The robot goes forward 20 in. The robot stops. The robot turns right 90 degrees. The robot goes forward 12 in. The robot stops. The robot turns left 90 degrees. The robot goes forward 44 in. The robot stops."


(a) Path from M1 to M8

(b) Path to M3

(c) Path from M3 to M7

Fig. 10. Case study tasks

5 Future Work

There are two main directions in which we intend to extend this work. First, we found that some input sentences may be invalid as they contain unclear or unknown information, meaning that they cannot be translated into robot functions even by a human, e.g. *"The robot hates moving forward"*. In order to make the system more robust to invalid inputs, it will be necessary to validate inputs with domain-specific formal reasoning and analysis. This will ensure the correctness of the system's understanding of the users' intentions and the correctness of the generated programs. Second, the function space contains several basic robot motions. As such, we intend to expand the function space with more low-level and even middle-level NL texts to develop the high-level abstraction self-explaining architecture further.

6 Conclusion

This work investigates the interdisciplinary NLP and robot path navigation task of breaking down complex high-level robot pathfinding abstractions into low-level NL instructions that can be processed directly by a LEGO NLPr system. The system we propose utilizes an efficient information extraction method with a OOP-CNL language model that analyzes and validates the sentence components' semantic meanings and relations. The system also contains an error-checking component that evaluates the input sentences' validity, and can also serve as a starting point for developing formal analysis methods for NLPr. We demonstrated how robot pathfinding problems for 2D Manhattan graphs could be handled by transforming the complicated high-level robot abstractions into a sequence of low-level NL instructions using NLP techniques and the domain-specific function library. The experimental results show that existing NLPr systems can be adapted to produce executable code using generated low-level NL specifications due to the simplicity, concreteness, and precise nature of the generated low-level sentences.

Although the study in this paper is limited in scope to pathfinding for LEGO Mindstorms EV3 robots, it lays a foundation for the task-driven HL2LL NL text self-explaining mechanism based on a domain-specific library. As complicated robot procedures can be explained using detailed sequential steps in natural language, we believe such a self-explaining mechanism could be a highly promising avenue for future NLP research.

References

1. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: learning to write programs. CoRR abs/1611.01989 (2016)
2. Dijkstra, E.W.: On the foolishness of "natural language programming". In: Bauer, F.L., et al. (eds.) Program Construction. LNCS, vol. 69, pp. 51–53. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0014656
3. Ernst, M.D.: Natural language is a programming language: applying natural language processing to software development. In: The 2nd Summit on Advances in Programming Languages, SNAPL 2017, CA, USA, pp. 4:1–4:14. Asilomar, May 2017

4. Hayes, B., Shah, J.A.: Improving robot controller transparency through autonomous policy explanation. In: Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction, HRI 2017, New York, NY, USA, pp. 303–312. Association for Computing Machinery (2017)
5. Hsiao, M.S.: Automated program synthesis from object-oriented natural language for computer games. In: Controlled Natural Language - Proceedings of the Sixth International Workshop, CNL 2018, Maynooth, Co., Kildare, Ireland, 27–28, August 2018, pp. 71–74 (2018)
6. Jurafsky, D., Martin, J.H.: Speech and Language Processing, 2nd edn. Prentice-Hall Inc., Hoboken (2009)
7. Kuhn, T.: A survey and classification of controlled natural languages. Comput. Linguist. **40**(1), 121–170 (2014)
8. Lauria, S., Bugmann, G., Kyriacou, T., Klein, E.: Mobile robot programming using natural language. Robot. Auton. Syst. **38**(3), 171–181 (2002). Advances in Robot Skill Learning
9. Lee, C.Y.: An algorithm for path connections and its applications. IRE Trans. Electron. Comput. **EC-10**(3), 346–365 (1961)
10. Lin, X.V., Wang, C., Pang, D., Vu, K., Zettlemoyer, L., Ernst, M.D.: Program synthesis from natural language using recurrent neural networks. Technical report, UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar 2017
11. Liu, H.: Metafor: visualizing stories as code. In: 10th International Conference on Intelligent User Interfaces, pp. 305–307. ACM Press (2005)
12. Loper, E., Bird, S.: NLTK: the natural language toolkit. In: Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics, ETMTNLP 2002, USA, vol. 1, pp. 63–70. Association for Computational Linguistics (2002)
13. Lopes, L.S., Teixeira, A.: Human-robot interaction through spoken language dialogue. In: Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No. 00CH37113), vol. 1, pp. 528–534 (2000)
14. Manning, C.D., Schütze, H.: Foundations of Statistical Natural Language Processing. MIT Press, Cambridge (1999)
15. Matuszek, C., Herbst, E., Zettlemoyer, L., Fox, D.: Learning to parse natural language commands to a robot control system. In: Desai, J., Dudek, G., Khatib, O., Kumar, V. (eds.) Experimental Robotics, pp. 403–415. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-319-00065-7_28
16. Maxemchuk, N.: Routing in the Manhattan street network. IEEE Trans. Commun. **35**(5), 503–512 (1987)
17. Menon, A.K., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.T.: A machine learning framework for programming by example. In: Proceedings of the 30th International Conference on International Conference on Machine Learning, ICML 2013, vol. 28, pp. I-187–I-195. JMLR.org (2013)
18. Mihalcea, R., Liu, H., Lieberman, H.: NLP (Natural Language Processing) for NLP (Natural Language Programming). In: Gelbukh, A. (ed.) CICLing 2006. LNCS, vol. 3878, pp. 319–330. Springer, Heidelberg (2006). https://doi.org/10.1007/11671299_34
19. Perera, V., Selveraj, S.P., Rosenthal, S., Veloso, M.: Dynamic generation and refinement of robot verbalization. In: 2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN), pp. 212–218, August 2016

20. Rangra, R., Madhusudan: Natural language parsing: using finite state automata. In: 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), pp. 456–463 (2016)
21. Rosenthal, S., Selvaraj, S.P., Veloso, M.: Verbalization: narration of autonomous robot experience. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, pp. 862–868. AAAI Press (2016). http://dl.acm.org/citation.cfm?id=3060621.3060741
22. Toutanova, K., Klein, D., Manning, C.D., Singer, Y.: Feature-rich part-of-speech tagging with a cyclic dependency network. In: Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, pp. 252–259 (2003). https://www.aclweb.org/anthology/N03-1033
23. Wang, X., et al.: Reinforced cross-modal matching and self-supervised imitation learning for vision-language navigation. CoRR abs/1811.10092 (2018)
24. Zhan, Y., Hsiao, M.S.: A natural language programming application for Lego Mindstorms EV3. In: 2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR), pp. 27–34, December 2018
25. Zhou, Y., Wang, W., He, D., Wang, Z.: A fewest-turn-and-shortest path algorithm based on breadth-first search. Geo-spatial Inf. Sci. **17**(4), 201–207 (2014)