Formal Validation for Natural Language Programming using Hierarchical Finite State Automata

Yue Zhan and Michael S. Hsiao
Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24060, U.S.A.

Keywords: Natural Language Programming, Natural Language Processing, Formal Validation, Hierarchical Finite State Automata.

Abstract: Natural language programming (NLPr) is a sub-field of natural language processing (NLP) that provides a bridge between natural languages (NL) and programming languages (PL), allowing users to design programs in the form of structured NL documents. Due to the imprecise and ambiguous nature of NL, it is essential to ensure the correctness of translation for critical applications where errors are unacceptable. Machine learning-based approaches for error checking are insufficient as it can be difficult for even the most sophisticated models to capture all the relevant intricacies of a natural language. Automata offer a formalism that has been used in compiling programming languages, and this paper extends automata-based methods to validating programs written in natural languages. In particular, we propose a hierarchically structured finite-state automaton, modeled based on domain-specific knowledge, for NLPr input validation and semantic error reporting. Experimental results from validating a set of collected NL sentences show that the proposed validation and error reporting can catch the unexpected input components while validating the semantics.

1 INTRODUCTION

Natural languages (NL) serve as the primary exchange of information among humans. Increasingly, NL is also becoming a key interface in the field of human-computer interaction. Natural language programming, an approach in which a computer program is constructed from NL, holds exciting promises of lowering the barrier to entry for programming and enabling new forms of interaction with devices.

Designing a system that understands NL and generates the corresponding code is challenging due to the massive amount of language features that the system must be able to interpret correctly. For example, users might give an NLPr system an input containing lexicon, grammar, or sentence structure that the system was not intended to handle or accept. Statistical n-gram language models are popular in the field of *information extraction* (IE) because of their forgiveness and robustness compared to rule-based grammars. However, such forgiveness might cause the loss of some semantics, thus making the NLPr system brittle (Kaiser et al., 1999). For example, in an n-gram based NLPr system where forgiveness and robustness are achieved by dropping unknown words in the input and where the words "love" and "hate" are unknown, the three sentences below will all result in the same output program. This treatment is undesirable to a human as the three inputs clearly have different meanings.

The robot loves going forward.
The robot hates going forward.
The robot goes forward.

One way to avoid translating with errors is to ensure that the NLPr system is able to accept, or reject, any sentence fully it receives as input, as opposed to statistically process the sentence as in machine learning based systems. Accepting a sentence can be performed by using a formal validation method to screen out invalid inputs. Such a formal validation method must detect unknown words/phrases, missing information, as well as words and phrases that are not compatible with the problem domain of the target programs. Whenever an invalid sentence is detected, the formal validation engine should provide feedback, and "debug" suggestions to help users fix their programs.

Finite-state machines (FSM) have played a significant role in both traditional and modern natural language processing (NLP) applications such as IE and natural language parsing (Manning and Schütze, 1999). In language processing, an FSM is an abstract machine with a finite number of states where the transition from one state to another is made according to a predetermined set of coded instructions as a sequential transducer (Rangra and Madhusudan, 2016). At each state, the next state is determined by the next input token in the sequence to select a relevant FSM path. FSMs serve as a powerful formal validation mechanism due to their deterministic properties, generating robust semantic representations that can lead to less error-prone results for NLPr systems. The correctness of the input processing and translation is critical for NL-based robot program synthesis. In addition, in the event that a user inputs NL sentences that the system is unable to process, we need to provide users useful and easy-to-understand feedback in order for the users to rectify their inputs. For example, in Figure 1, the token "randomly" brings the system to the error state from S_3 because the word is not covered in the system's lexicon. If the system simply terminated at this line without providing any error messages, it would leave the users no clue about how to fix the problem. In fact, an NLPr system that does not provide useful feedback is difficult to use for non-expert users, negating much of the appeal of NLPr.

In order to validate input sentences and report possible errors, the system must not only parse the sentences into their constituent words, necessary for generating expressive language intermediate representations for downstream processing. It must also identify unknown words and phrases in sentences that are not covered in the language model, thus avoiding possible misinterpretations. In this paper, we address the challenge of semantic parsing and input sentence validation by implementing a context-sensitive analysis engine. An automaton can be used to validate an NL sentence by checking the end state reached by the sentence. If the *accept* state is reached at the end of the sentence, then the sentence is valid. On the other hand, if the automate ends in the *error* state, then the sentence contains unknown or unexpected language components or missing some of the expected information. Different error messages can be generated depending on exactly how the *error* state was reached.

However, modeling all language features into one modular FSM-based system is a complicated and challenging process, resulting in a hard-to-maintain system. A hierarchically structured machine can reduce the complexity of the system by breaking the state machine into several *superstates*, denoted as SS_i in this paper, where $SS_i \in \mathbb{SS}, 1 \leq i \leq m$, as shown in Figure 2. A *superstate* represents a cluster of one or

[1] Stands for "end of sentence"

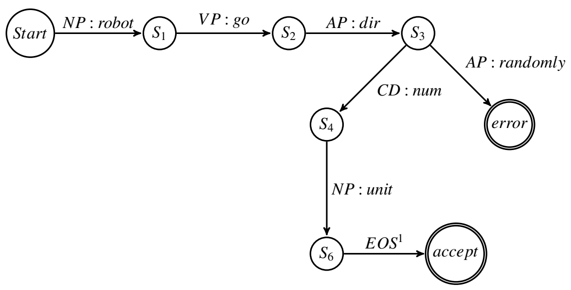

Figure 1: Example FSM.

more substates, noted as S_x, where $S_x \in \mathbb{S}, 1 \leq x \leq n$, as in Figure 1. Such a structure makes it possible to view the validation system at different levels of abstraction, making the system easier to reason about and makes it possible to specify the system more in detail (Alur and Yannakakis, 1998). Such a hierarchical structure also simplifies the addition of new language features to the model, allowing new features to be added without modifying all other states' previous transition conditions. The hierarchical structure also allows us to reuse states when transitioning between the super states. For example, as shown in Figure 4, the sensor-related states are reused by the robot *superstate* SS_1. Finally, although hierarchical FSMs have played important roles in handling non-terminal nodes in a *Context-Free Grammar* (CFG), they cannot be easily extended to handle ambiguity in a language. In this paper, we present a hierarchical FSM for formal validation of inputs and error reporting for the task of robot navigation using NLPr.

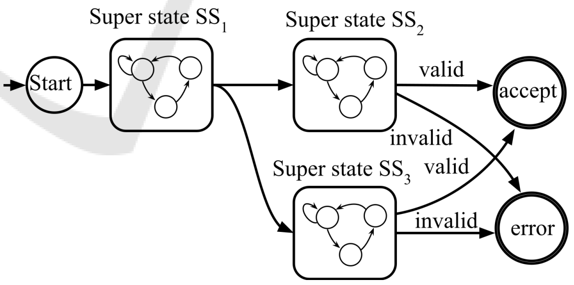

Figure 2: Example HFSM.

2 RELATED WORK

Finite-state automata are widely used in several statistical NLP tasks such as lexical analysis, tokenizing, part-of-speech (POS) tagging, and sentence parsing. In (Rangra and Madhusudan, 2016), the authors proposed a Modular Finite State Automa (MFSA) for sentence grammar validation based on POS. Their MFSA recognized the validity of token combinations in the sentence and built the parse tree in a bottomup manner after completing all modular validations. This approach reduced repeat parsing by using submodules based on grammatical categories. However, this MFSA did not address issues like ambiguity resolution.

An FSM-based parser application is presented in (Graham and Johnson, 2014) for parsing Internet protocols. The authors claim that FSM approaches are straightforward to construct and maintain and that they are sufficiently expressive for high-throughput applications like Internet protocols. This stands in contrast to the previously dominant belief that FSMs are not scalable or expressive and that they take too much time actually to implement. This work demonstrated that the integration of domain-specific knowledge into an FSM is feasible.

A hybrid FSM framework that combines the benefit of spell checking and machine translation approaches is presented in (Beaufort et al., 2010). It performs NL SMS text normalization, converting noisy SMS conversations to a more standardized form. The major drawback of training the model using SMS text is that building a machine that can transcribe the corpora to standard written forms is difficult due to the significant variation present in SMS conversations. As such, the authors stated that future works using similar approaches should be based on text with a more rigid structure than SMS messages.

Finite state-based compiler toolkits like Foma (Hulden, 2009) and OpenFst (Allauzen et al., 2007) also embrace the formal analysis power of statistical finite state techniques.

Finite-state methods are also leveraged in the fields of machine learning and Neural Machine Translation (NMT). A deterministic pushdown transducer (DPDT) is adopted in (Moisl, 1992) for general NLP, and a simple recurrent neural network is trained to simulate a finite state automaton. The authors demonstrate that using such NN-based implementation results in a system that can meet a typical NLP system's desirable properties for language parsing.

In (Goyal et al., 2016), neural language models are used to regularize finite state machines with a character-level model for NL text generation. The weighted finite-state automaton proposed in this paper incorporates prior knowledge to guide an RNN to generate valid and well-formed character sequences.

In the works (Stahlberg et al., 2019)(Yuan et al., 2019), the authors use finite-state transducers that are built from unlabeled corpus to constrain the output of a neural beam decoder to solving Grammatical Error Correction (GEC) tasks in NLP. They claim that these methods can also be used with statistical machine translation (SMT) approaches if training data is available, and they report superior gains over SMT baselines. Their work suggests that finite state techniques can enrich the search space of possible grammatical errors and corrections and constrain the neural GEC system.

3 THE PROPOSED CONTEXT-SENSITIVE HIERARCHICAL FSM (HFSM)

Our approach to semantic parsing is to transform an NL input into an intermediate semantic representation with domain-specific information. Semantic parsing for general NL inputs can be difficult due to the underlying text's imprecision and ambiguity. In a LEGO robot NLPr application (Zhan and Hsiao, 2018)(Zhan and Hsiao, 2020), the lexicon and the domain-specific function library $\mathbb{F}$ is built upon the functionalities allowed by the hardware and sentences manually collected from users that describe robot movements. We restrict the application domain to be the LEGO EV3 robot and constrain the language to be a semi-controlled natural language (CNL) (Kuhn, 2014). By restricting the NLPr system's problem domain to LEGO robot program synthesis, the problem of semantic parsing can be modeled into a domain-specific context-sensitive FSM. Such a machine performs semantic checking and validation on a subset of NL, e.g., a semi-controlled natural language. The object-oriented language style also helps to reduce ambiguity in sentences, as each phrase must involve an object.

Unlike conventional programming languages that are precise and unambiguous by design, we must deal with NL inputs that could be imprecise and ambiguous. Parsing an NL sentence into the grammatical form using a grammar that ignores context, such as CFG, might cause the loss of semantic information and other problems. Even though modeling all language features into one single NLPr system is impractical, if not impossible, the task can be made easier by restricting the lexicon and language features to a finite size, such as the lexicon describing robot tasks. With this domain-specific knowledge-based lexicon and library, the ambiguity and imprecision in the input NL texts can be mitigated. Therefore, instead of fully encoding the NL sentences into an abstract form, some particular categories are kept and mapped based on the library, while others may be dropped. Words that the system does not care about, such as "a", "the", "that" are dropped from the input. For example, the sentence "*The robot turns on the LED in* *red*" is pre-processed to "*robot turns on LED PREPOSITION RED.*", keeping the sentence structure as well as the subjects and properties of the action, such as the color RED in this example.

3.1 Construction of the HFSM

In order to validate NL texts fed into the NLPr system and detect sentence components, we propose a hierarchical FSM with a semantic checking mechanism. The hierarchical FSM is constructed based on basic English grammar, the domain-specific lexicon, and libraries of the LEGO robot application. The FSMs are developed based on a set of NL sentences that describe the capabilities of the robot. The first *superstate* handles the initial transitions to other *superstates* by detecting the objects and actions in a sequence of word tokens. We define nouns, verbs, and prepositions based on POS tagging of the input sentences, using the Penn Treebank standard (Taylor et al., 2003), and denote the following word classes:

- NP: Noun and noun phrases. For example: the robot, sensors, LED, servo, etc.
- VP: Verb. For example: turn, move, delay, etc.
- AP: Adjective, Adverb. For example: colors, directions, etc.
- PP: Preposition, preposition phrases. For example: in, on, from, to, etc.

There are some exceptions to these classes, including the following: the customized object names are treated as NP, declared variable names are treated as NP. For example, the *happy* in the sample sentence "*The robot is happy.*" is treated as a Boolean variable. The exception treatment treats random variable names as NP as well, as in sentences such as "*The robot is variable_xyz.*". Grammatical faults like this are forgiven in the FSM-based parsing system. Each FSM has one start state and two terminal states: *accept* and *error*.

In the figures, circles represent states, denoted as S_x, and rounded rectangles represent *superstates* denoted as SS_i. Arrows denote the transition from one state to another, with the label denoting the token that triggered that transition, as shown in Figure 3. The ordering of tokens in the input holds important information about the sentence's meaning, and the HFSM can account for this. Depending on the ordering of tokens in a sentence, the HFSM may transition between *superstates* in a different order, thus parsing the context. For example, in the simplified HFSM in Figure 3, the combination of the NP `robot` and VP `see` indicates the usage of the color sensor or the ultrasonic sensor, and as such the HFSM transitions from the robot superstate SS_1 to the sensor superstate SS_2. The combination of the NP `robot`, NP `LED` and VP `turn` indicates the LED function. Thus, the HFSM transits from SS_1 to the LED superstate SS_3. The hierarchical structure of the HFSM allows us to model the system by building individual sub-FSMs based on the domain-specific function library for the LEGO robot and also improves maintainability and scalability.

In addition to handling simple action statements, the LEGO NLPr system also handles Condition-Action statements with flexible sentence structure. Logical keywords such as `if`, `else`, `while`, `and`, and `or` are treated similarly to how they are in standard computer programming languages.

The same token sequence can correspond to different desired functions depending on whether the sequence is part of a condition or an action. For example, the token sequence in the conditional statement `if` "the robot is happy" refers to a Boolean variable checking function to see if the *happy* is `True`, while the same token sequence in the action statement "the robot is happy" refers to a variable assignment function that assigns *True* to `happy`. In order to account for this behavior and correctly parse and validate the semantic meaning of the input sentences, the main FSM is split into two sub-machines: condition FSM and action FSM. Each sub machine part has its own properties of validation.

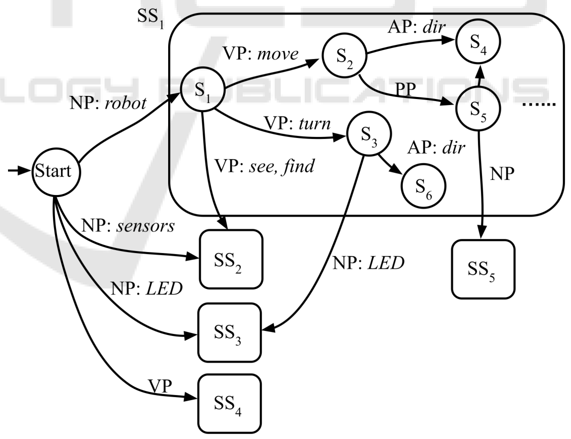

Figure 3: Example HFSM with some states omitted.

3.1.1 The Condition FSM

The LEGO NLPr system accepts conditional statements of the following types: 1) sensor usage: checking the value read from a sensor. There are four different sensors available on the LEGO Mindstorms EV3 platform: gyro sensor, touch sensor, ultrasound sensor, and a color sensor. 2) variable usage: checking the value of a variable. A variable can be a Boolean, a numeric value, or a string. However, all variable names must be declared in advance. Using a variable without first initializing it results in the HFSM transitioning to the *error* state. For example, checking *"If the robot is happy, ..."* will result in the HFSM reaching the *error* state and raising an error about the undeclared variable `happy` if there is not a statement such as *"The robot is happy."* ahead of it.

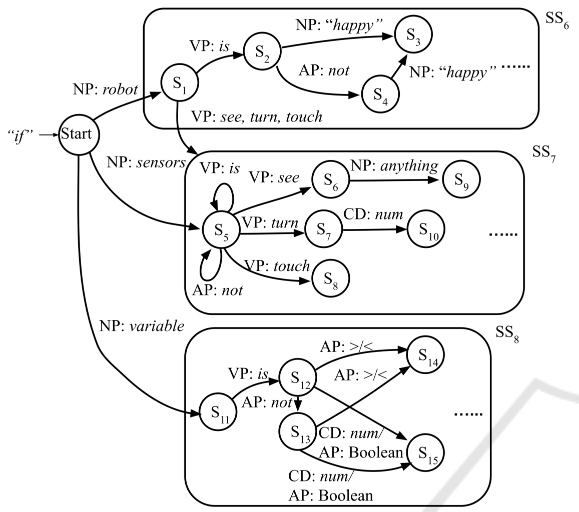

Figure 4: Condition HFSM with some states omitted.

3.1.2 The Action FSM

The LEGO hardware limits the action space that the robot operates in. The functions in the LEGO domain-specific function library can be split into multiple sub-FSMs based on the primary object: 1) the robot, 2) the sensors, 3) the LED, 4) default attachments that are always connected to the robot, such as the servo and 5) new numeric and Boolean variables, as well as new custom user-defined robot attachments, which need to be declared in advance.

3.1.3 Implementing Context-sensitivity

Unlike CFG, the proposed HFSM considers a token's context when evaluating its validity, enabling it to detect errors in the sentences' semantic meaning early. The HFSM's sensitivity to context is also different from simple sentence pattern-matching because our approach reports errors in semantics.

In order to ensure the correctness of the NLPr system's understanding of how to map inputs to their corresponding output programs, context analysis is required to guarantee that context-sensitive requests are handled properly.

To achieve such context analysis, context-sensitive rules γ applied in the HFSM are modeled based on the domain-specific function library of LEGO robot. Context sensitivity helps catch errors that arise in situations where input is syntactically valid but not semantically valid such as ambiguous inputs, undeclared variables, mismatched hardware functionalities, and unimplementable or prohibited behaviors. For example, the sentence "The robot moves forward 90 degrees." would pass syntactic checking, as shown in the *Superstate* SS_1 in Figure 6. However, the combination of the action `robot.move(forward)` and the unit `degree` poses a problem for an NLPr system as the `move` function expects a distance variable but receives an angle instead. Such semantic checking is done by applying the context-sensitive rules γ after input token sequence validation has been performed.

After the validation of the input token sequence, and before making a final assertion of the input's validity, a case analysis for context-sensitive semantic checking checks if the input satisfies a given property P is applied as the final check. The case analysis acts as a checklist with multiple registers that need to be set to pass validation. Take the LED *superstate* SS_3 as another example. When the sequential tokens pass the structure validation, instead of a directly transitioning to the *accept* or *error* states, there is a semantic checking multiplexer by the end of the LED *superstate* SS_3, shown in Figure 7. The multiplexer checks if the information extracted from the sentence can form valid token combinations that the NLPr system can then use to generate corresponding code. For example, if the target LED color and the original LED color are the same, which makes no sense in terms of controlling the robot, the input will be marked as invalid, as in the invalid sentence "*The red LED turns red.*" If a target LED color, such as blue, is not supported, this would result in an error as well, as shown in the invalid sentence "*The LED turns on in blue.*"

A similar approach is used to detect sentences with run-on errors as well. For example, the checker checks the registers for the target LED color and the original LED color to detect the error in the sentence "The LED turns from red to green from red to green." If one color register is already set when assigning a new value, we know it could be a possible run-on sentence with redundant or dangling terms.

3.2 The Error Reporting Mechanism

Much research on engineering education has highlighted the importance of high-quality and easily understandable error messages for beginner students for learning a programming language (Traver, 2010)(Munson and Schilling, 2016)(Crestani and Sperber, 2010)(Marceau et al., 2011). This is simi-

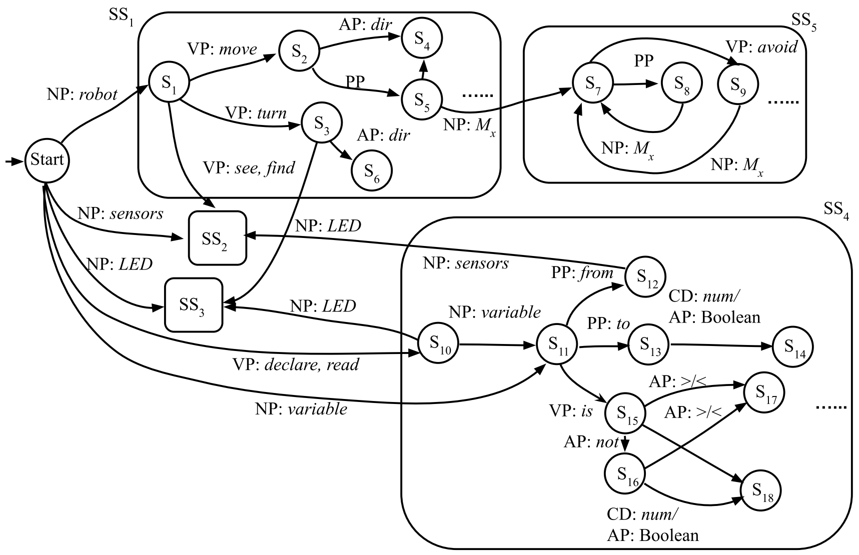

Figure 5: Simplified action HFSM with some states, including terminal states, omitted.

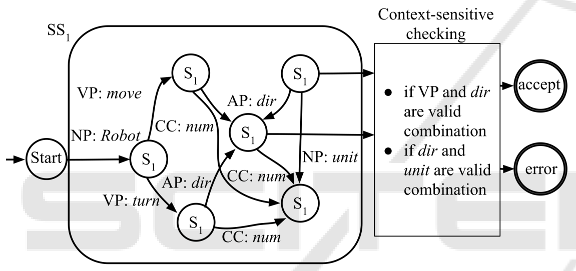
SS1
Context-sensitive checking
Start
NP: Robot
VP: move
VP: turn
CC: num
AP: dir
AP: dir
CC: num
NP: unit
CC: num
S1
if VP and dir are valid combination
if dir and unit are valid combination
accept
error

Figure 6: Simplified robot *Superstate* SS_1 with basic motions related states only.

larly important for debugging natural language sentences, such as in GamgChangineer, an NLPr application for video game design (Hsiao, 2018). The phrasing and quality of error messages play a fundamental role in how feedback mechanisms impact students' actions when wrestling with untranslated NL sentences, as well as the human-computer interaction between the NLPr system and the users.

NL parsing is the process of transforming raw NL inputs into a more standard and meaningful form that can be understood and processed by the NLPr system. By transforming and validating the inputs to the NLPr system, the FSM-based parser *imbues the input sentences with trust* (Graham and Johnson, 2014), which means that we expect the sentences input to the NLPr system to be written using somewhat reasonable standard forms and structures.

As the NLPr system expects input sentences to be well-formed, we augment the FSM-based validation to include an error reporting framework that helps guide users towards creating valid inputs. This FSM-based error reporting framework leverages domain-specific knowledge and lexicon for the NLPr application. The error reporting functionality of this framework provides early detection of not-covered, unimplementable, untranslatable, and ambiguous language components in the inputs. Error and warning reporting alone does not provide much information for beginner users. Therefore, in addition to meaningful error messages, example sentences related to the sentences' semantic meanings are provided in accordance with the *teaching by example* paradigm.

The errors that challenge the NLPr system can be categorized into three main types:

1. **Not-covered Language Components.** These errors mean that there are unknown words or unexpected sentence structures in the input. For example, the sentence "*The robot goes forward randomly.*" shown in Figure 1 is invalid because the token "randomly" is not covered in the lexicon. Thus, the error message generated would point out that "randomly" is not understood by the system and that it will be ignored when generating the final program.
2. **Missing Information.** These errors mean that the information extracted from the sentence is not sufficient for function matching or program synthesis. For example, while the sentence "*The robot goes forward.*" is grammatically correct, it is not concrete enough for generating executable programs for a robot as it is missing details such as how far the robot should move forward. In a situation such as this, where information is missing, default values will be used substituted. There are many other forms such these errors may take, such as the sentence "*Read variable_xyz from the*

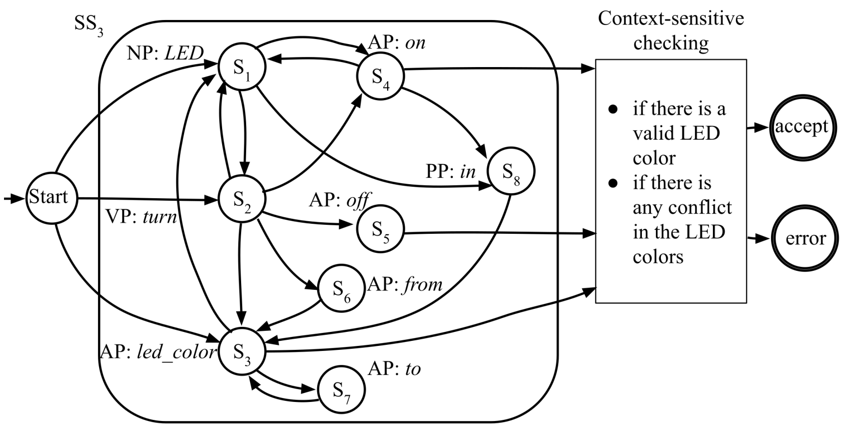
SS3
NP: LED
S1
AP: on
S4
Context-sensitive checking
• if there is a valid LED color
• if there is any conflict in the LED colors
accept
error
Start
VP: turn
S2
PP: in
S8
AP: off
S5
S6
AP: from
AP: led_color
S3
AP: to
S7

Figure 7: Simplified LED *Superstate* SS_3 with some states omitted.

sensor", which does not adequately specify which sensor is being used.

3. **Mismatched Information.** These errors mean that the extracted sentence contains conflicting information that cannot map to a unique match in the function library. For example, the object `color sensor` and the action `touch` together would result in a transition to an error state as there is no valid function that can both handle color and touch. Function properties also need to be validated when a valid function is matched. For example, the unit *inch* does not fit the function `robot.turn(left)` in the sentence "*The robot turns left 90 inches.*"

Currently, the error message library contains 118 distinct error and warning message templates associated with LEGO robot objects and actions. Each message has a unique error ID. Error messages describe precisely what part of the input caused the validation error, whether that be a problematic token, a missing token, or a mismatch between an object and its action. Many error messages are accompanied by handwritten examples of proper inputs, which serve to show users how to correct their errors directly.

One sentence can result in multiple errors. For example, the sentence in Example 1 results in three error messages. Error #117 points out that the token "*avoid*" is not understood by the system. Error #13 and error #16 are generated because the system parses the sentence in as "*robot move forward*", but no information is given that dictates how far the robot should move forward.

Not all errors result in the termination of the translation process for the given sentence. For some errors, such as certain missing information errors, the system can still generate output code by inserting default values. For example, the token "*avoids*" in Example 1 would be skipped, and the rest of the sentence will be filled with default values in order to generate the program, as seen in error #13 and error #16.

Example 1. *"The robot avoids moving forward."*
Error messages:

- *ERROR(117): Unknown word found when describing a robot action. Please consider rewriting the sentence without it. In the LEGO NLPr system, you need to describe what needs to happen step by step to avoid ambiguity. I see that the next word is an action. I will skip the current word, which might cause mistranslation: *avoid**
- *ERROR(16): It seems like we are missing a number. Please write numbers in numerical digits 0 9. A default value will be used.*
- *ERROR(13): It seems like we are missing a unit. You can turn for "degrees", move for "inches"/"cm"/"seconds", or stop for "seconds". I will use default now.*

However, the system is not able to handle all possible cases where an unknown token is in the input. For example, the sentence in Example 2 cannot be translated because the object "right arm" is not recognized by the system, and if it were simply dropped, then no action would take place.

Example 2. *"Turn the right arm up."*
Error messages:

- *ERROR(24): It seems like we are missing an object or a sensor that can be recognized. Please check your spelling or re-write the sentence in an object-orientated way.* ***For example****, use "the robot turns left 90 degrees." instead of "turn the robot left 90 degrees."*
- *ERROR(48): Possible undefined variable or Boolean status. It looks like you are checking a variable or status that's not previous defined. You*

can declare a variable by using ***example sentences*** *like "*xyz* is 5.", "*happy* is True", "set *distance* to 10 inches." or simply use "xyz = 5"(do not forget the space). Can you check the object in: *right arm up**

- *ERROR(80): Unable to understand this action. Possible sentence fragment!*

For NL sentences with multiple conditions or multiple actions, the sentences are first split into condition set and action set and then processed separately. For example, the conditional statement "*if the distance is larger than 10 inches and the distance is less than 50 inches, ...*" contains two conditions: `distance>10` and `distance<50`. This approach allows the system to treat each condition independently. As such, the valid parts can be translated even if some of the conditions are not understood by the system. For example, in the condition statement "*If the robot is happy or the robot is going forward, ...*", the first condition `happy==True` is valid while the second condition raises the following error in Example 3.

Example 3. *Condition: "robot is going forward "*
Error messages:

- *ERROR(118): Can't check the moving direction as a condition. Please consider declaring a robot status.* ***For example****, "If CONDITION, the robot goes forward 10 inches." then use the same CONDITION for the desired action.*

4 EXPERIMENTAL RESULTS

To evaluate the performance and the coverage criterion of the proposed HFSM-based semantic parsing and error-reporting system, we used two sets of input sentences: 1) 630 total valid and invalid sentences collected manually, 2) 1148 stochastically generated sentences. These sentences are randomly synthesized using a Markov chain based method based on the manually collected sentences and the domain-specific lexicon. The HFSM-based formal validation helps to effectively reduce the mistranslations caused by simply accepting a sentence based on the n-gram based models keyword matching.

4.1 Manually Collected Sentences

The manually collected and annotated test set covers each transition of the HFSM and each nested substate at least once. 422 out of 630 sentences are valid by default and pass the validation through the proposed HFSM. 3 *false negative* cases and 7 *false positive* cases are in the rest 208 ill-designed sentences, which results in a *precision* of 98.21% and a *recall* of 99.23%. 892 error messages are generated for the 208 erroneous sentences, minimum 1 message, maximum 8 messages, and mean 1.38 messages. The formal validation system correctly classifies 98.41% of the manually collected sentences as valid or invalid.

4.2 Auto-generated Sentences

A simple Markov chain-based NL text generator is implemented to generate a rich set of testing sentences. 1148 sentences with a max sentence length of 30 words are produced. Because of the stochastic nature of the Markov chain model, the sentences generated are random and error-prone. This is desirable since we want to see if the corresponding error messages can provide helpful information for the users. 290 out of the 1148 sentences generated are semantically and grammatically correct and are successfully validated by the HFSM. After running the rest of the sentences through the HFSM, 50 *false negative* cases and 15 *false positive* cases are identified, which results in a *precision* of 94.77% and a *recall* of 84,47%. 1619 error messages are generated for the 858 erroneous sentences, minimum 1 message, maximum 7 messages, and mean 1.89 messages. The formal validation system achieves an accuracy of 94.34% on the auto-generated dataset. Therefore, the overall accuracy of the formal validation engine is 95.78%.

4.3 Case Study

In order to demonstrate how the semantic parsing and error reporting system works, some examples are discussed above and in this section. In Example 4, the condition "*if the touch sensor*" is clearly a sentence fragment, thus labeled with Error #9, #23 and #39.

Example 4. *"The robot picks up the robot if the touch sensor."*
Error messages:

- *ERROR(9): missing action for sensor condition.*
- *ERROR(23): incomplete sentence. Please check if you are missing an action word, numbers, units, colors, or directions.*
- *ERROR(39): sentence fragment found. In our system, the "and" and "or" are keywords used to parse the sentence.* ***For example****, a conditional statement can be written as "if CONDITION1 and CONDITION2, ACTION1 and ACTION2". Each CONDITION and ACTION will be processed separately. Therefore, to avoid ambiguity in conditions, instead of using sentences like "if the color sensor sees black or white, *action*", use "if the*

*color sensor sees black, *action*. Else if the color sensor sees black, *action*".*

- *ERROR(116): Unknown word found when describing a robot action. Please consider re-write the sentence without it. In the LEGO NLPr system, you need to describe what needs to happen step by step to avoid misunderstanding. Can you check what is: *pick**
- *ERROR(17): Expect an action word for the robot object.*

While evaluating the performance of the proposed HFSM using collected sample sentences, we also found corner cases where *false negatives* and *false positives* exist, which indicates that future development and improvement on the HFSM's structure are needed. Example 5 shows a *false negative*, an auto-generated sentence that passes validation despite being erroneous. This error occurs because the declare *superstate* SS_4 first skips the unknown word "*variable_xyz*" and then checks the rest of the sentence to see if the unknown word needs to be treated as a variable name or not. When the submachine passes the previous state to the next state after skipping the custom variable token "*variable_xyz*", the action "*set*" overwrites the first action `read`. Thus, the sentence is treated as "*The variable_xyz sets to 10 seconds.*"

Example 5. *"Read the variable_xyz sets to 10 seconds."*

The unknown word is initially ignored for variable declaration functions modeled in SS_4 because it is uncertain whether the unknown word is a variable name without checking the context of the rest of the sentence. When the rest of the input tokens indicates this sentence assigns variable value, the unknown word then is recognized as a variable and pass the validation. This kind of mistake could be fixed by incorporating short term memory or a 2-pass analysis.

A *false positive* example is shown in Example 6. While error #118 and error #17 are correctly raised by the condition and action in the sentence, error #116 regarding unknown word is falsely raised because it accidentally treats the end of sentence "." as a word token since the usage of punctuation in this circumstance is not considered while modeling the FSM.

Example 6. *"The robot if the robot goes to M1."*
Error messages:

- *ERROR(118): Cannot check the moving direction as a condition. Please consider declaring a robot status.* ***For example****, "If CONDITION, the robot goes forward 10 inches." then use the same CONDITION to for the desired action.*
- *ERROR(116): Unknown word found when describing a robot action. Please consider re-write the sentence without it. In the LEGO NLPr system, you need to describe what needs to happen step by step to avoid misunderstanding. Can you check what is: *.**
- *ERROR(17): Expect an action word for the robot object.*

5 CONCLUSION AND FUTURE WORK

We have presented an HFSM-based formal validation of NL inputs for the LEGO robot natural language programming application. This system processes input sentences using a hierarchical structured FSM to extract information for generating intermediate representations for valid sentences and meaningful error messages to help users "debug" invalid sentences. As such, the unknown and missing language components would not simply be ignored while generating executable programs for LEGO robots. The HFSM is modeled based on basic English grammar rules with the additional use of exceptions and domain-specific knowledge for the LEGO robot. The transitions between two states/superstates are context-sensitive, allowing the detection of any mismatching/unknown/missing information. This detection enriches the meaning of generated error messages and provides more useful error messages to users.

The use of hierarchical structure also eases the maintainability of the system and significantly reduces the number of repeat states needed. Since an input sentence will only be labeled as valid if the transitions result in a final success state, the process of translating natural language sentences to executable programs is more reliable. However, the system may still be tricked by some carefully crafted sentences and is subject to some restrictions in sentence structure due to the HFSM's preference for an object-oriented language style. Despite these limitations, it is well-suited for teaching novice users logical thinking and problem-solving skills.

Future work includes improving the modeling to include more language components and more diverse structure, expanding the error messages dictionary, and investigating the applicability of this technique to new problem domains.

REFERENCES

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). Openfst: A general and efficient weighted finite-state transducer library. In Holub, J.

and Žďárek, J., editors, *Implementation and Application of Automata*, pages 11–23, Berlin, Heidelberg. Springer Berlin Heidelberg.

Alur, R. and Yannakakis, M. (1998). Model checking of hierarchical state machines. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, page 175–188, New York, NY, USA. Association for Computing Machinery.

Beaufort, R., Roekhaut, S., Cougnon, L.-A., and Fairon, C. (2010). A hybrid rule/model-based finite-state framework for normalizing SMS messages. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 770–779, Uppsala, Sweden. Association for Computational Linguistics.

Crestani, M. and Sperber, M. (2010). Experience report: Growing programming languages for beginning students. ICFP '10, page 229–234, New York, NY, USA. Association for Computing Machinery.

Goyal, R., Dymetman, M., and Gaussier, E. (2016). Natural language generation through character-based RNNs with finite-state prior knowledge. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 1083–1092, Osaka, Japan. The COLING 2016 Organizing Committee.

Graham, R. D. and Johnson, P. C. (2014). Finite state machine parsing for internet protocols: Faster than you think. In *2014 IEEE Security and Privacy Workshops*, pages 185–190.

Hsiao, M. S. (2018). Automated program synthesis from object-oriented natural language for computer games. In *Proceedings of the Sixth International Workshop on Controlled Natural Language, August, 2018*, pages 71–74.

Hulden, M. (2009). Foma: A finite-state compiler and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*, EACL '09, page 29–32, USA. Association for Computational Linguistics.

Kaiser, E. C., Johnston, M., and Heeman, P. A. (1999). Profer: predictive, robust finite-state parsing for spoken language. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No.99CH36258)*, volume 2, pages 629–632 vol.2.

Kuhn, T. (2014). A survey and classification of controlled natural languages. *Comput. Linguist.*, 40(1):121–170.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011). Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, page 3–18, New York, NY, USA. Association for Computing Machinery.

Moisl, H. (1992). Connectionist finite state natural language processing. *Connection Science*, 4(2):67–91.

Munson, J. P. and Schilling, E. A. (2016). Analyzing novice programmers' response to compiler error messages. *J. Comput. Sci. Coll.*, 31(3):53–61.

Rangra, R. and Madhusudan (2016). Natural language parsing: Using finite state automata. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 456–463.

Stahlberg, F., Bryant, C., and Byrne, B. (2019). Neural grammatical error correction with finite state transducers. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4033–4039, Minneapolis, Minnesota. Association for Computational Linguistics.

Taylor, A., Marcus, M., and Santorini, B. (2003). *The Penn Treebank: An Overview*, pages 5–22. Springer Netherlands, Dordrecht.

Traver, V. J. (2010). On compiler error messages: What they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010.

Yuan, Z., Stahlberg, F., Rei, M., Byrne, B., and Yannakoudakis, H. (2019). Neural and FST-based approaches to grammatical error correction. In *Proceedings of the Fourteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 228–239, Florence, Italy. Association for Computational Linguistics.

Zhan, Y. and Hsiao, M. (2020). Breaking down high-level robot path-finding abstractions in natural language programming. In *NL4AI@AI*IA*.

Zhan, Y. and Hsiao, M. S. (2018). A natural language programming application for lego mindstorms ev3. In *2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, pages 27–34.