Multi-Phase Context Vectors for Generating Feedback for Natural-Language Based Programming

Michael S. Hsiao
Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA USA 24061

Abstract

Automatic generation of feedback messages in a natural-language based programming for video games is presented. The input sentences are processed in four stages. During each stage, context vectors are aggregated and any violation to a syntactic or semantic rule is reported to allow users to debug and fix the text. The results discuss a list of common errors detected by the proposed method.

1 Introduction

Programming in the user's native language attempts to directly convert instructional text to an executable program. The benefits of such a programming system are many, including increased productivity, reduced effort to learn conventional programming languages and debugging, etc. Thus, proficiency in a NL-based platform will help carry over to learning a conventional object-oriented programming language later on. However, programming in a NL faces many hurdles, including the resolution of ambiguity/imprecision, handling of incomplete sentences, and propagation of context from one sentence to the next. Rather than targeting general-purpose programming with NL, aiming for domain-specific applications should be the first goal. With a specific domain, we can narrow the scope for that target application, and the accepted language resembles somewhat to a controlled natural language (CNL), with a finite set of nouns, verbs, and phrasal structures. However, the grammatical rules used in this work are not as constrained as in most CNLs. Instead, the sentences do not need to conform to rigid grammatical structures.

Motivating Example: Alice wishes to write a program involving a rabbit, fox, and carrots. She writes: *"There are 3 foxes, 20 carrots, and a rabbit. The rabbit moves around. When a rabbit touches a carrot, it eats the carrot. When the rabbit sees a fox, it chases it."*

Such a programming paradigm is much more natural to those who have little experience writing a program, and the users can play the resulting game, providing a positive feedback. Moreover, fixing errors in NL offers an early introduction to debugging. For example, consider the last sentence in the above example: “*When the rabbit sees a fox, it chases it.*” There are multiple possible interpretations for the phrase “it chases it”, and the system should be able to offer feedback to the user about this potential bug.

A platform has been constructed for this purpose to create video games. The user enters the program that describes the logic of the game in English. The text is then translated to an executable, playable game via a 4-stage compilation process: syntactic processing, phrasal semantic processing, sentential semantic processing, and code generation. At each stage, a context vector is produced and aggregated. Analyses of the context vectors against syntactic and semantic rules help to generate error messages that pinpoint any imprecise, ambiguous, and/or incorrect expressions. The user can then use these error messages and suggestions to fine-tune and debug their NL text. Analysis of the games written by middle-school students show a list of common errors captured by the system.

The rest of the paper is organized as follows. Section 2 provides the preliminaries and background. Section 3 details the methodology for generating the context vectors and error messages based on these vectors. Section 4 discusses the results and Section 5 concludes the paper.

2 Preliminaries

Let **W** be the sequence of words, $w_0, \ldots, w_n$, in a sentence; each word in a valid sentence should be able to be mapped to a valid token during first step of parsing. The categories for any valid token is ***L*** *= (E, A, T, P, S)*, where E: the set of entities (or objects), A: the set of actions, T: the set of attributes, P: the set of predicates, and finally, S: the set of optional selectors. Note that all these sets in ***L*** can grow and evolve with time.

For the domain of video games, the set of entities, *E*, is the set of characters involved in the game, such as foxes, rabbits, etc. The set of actions, *A*, may include chase, flee, wander, jump, die, etc. Third, the set of attributes, *T*, includes the color, speed, etc. associated with the characters. Note that the user can add more attributes on the fly. Next, the set of predicates, *P*, may include see(), reach(), touch(), catch(), etc. Finally, the set of selectors, S, allows the user to say something like "when 35 rabbits are gone".

Note that new terms can be learned in *T*. For example, the sentence "*When a fox sees a rabbit, it becomes happy. When a fox is happy, it …*" The term 'happy' is learned and associated with the behavior at run-time, as explained in Hsiao (2018).

In Hsiao (2018), error reporting was limited. Later, in Zhan & Hsiao (2019), an attempt to use machine learning to categorize types of errors was made, again with only limited success. Notably, a small change in a sentence may result in completely different type of error. Thus, accurately mapping an erroneous sentence to a specific error (among a potentially large number of errors) via machine learning alone is likely infeasible. Instead, rules can more accurately capture the formal relations in the context of a sentence that imply an error. In other words, analyses on the aggregated context vector against a rule set can generate the needed error message(s) accurately.

3 Methodology

The four stages for the compilation process is illustrated in Figure 1. The key to our approach is that each stage works on a distinct level of abstract representation of the input text. Context vectors were custom designed for the game domain. However, the context vectors can be tailored according to the needs of a domain. Example fields of the context will be described within each stage.

Stage 1: Syntactic Analysis

Given the sequence of *n* words, $w_0, \ldots, w_n$, this stage aims to produce a sequence of *m* tokens, **T**, $t_0, \ldots, t_m$, where $m \leq n$, and a syntactic context vector, **SC**. Every token takes a type as defined in ***L*** explained earlier, such as character, verb, predicate, attribute, etc. Any typo (no match to any word in the lexicon) or grammatical error (such as 'fox chase' instead of 'fox chases') will be output to the user during this stage as well.

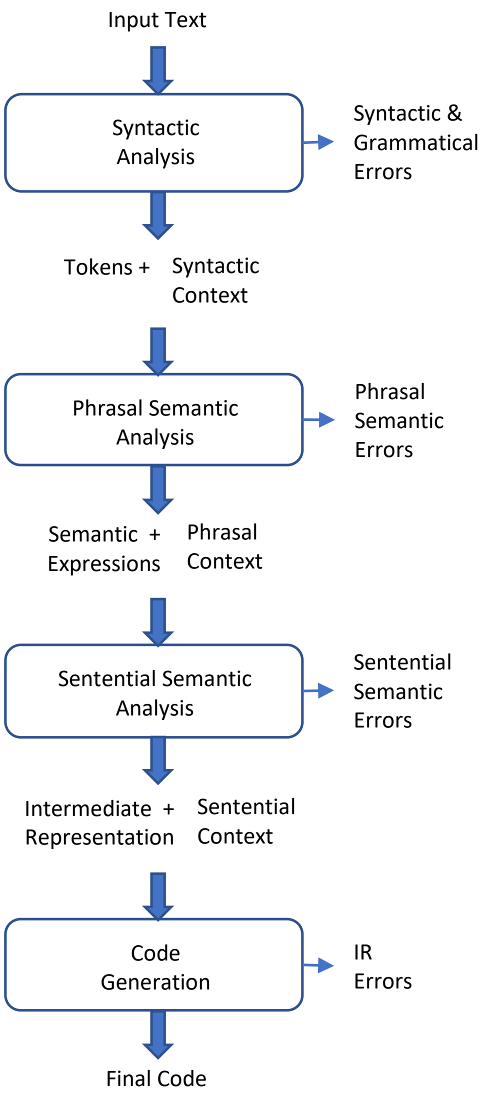
Input Text
Syntactic Analysis
Syntactic & Grammatical Errors
Tokens + Syntactic Context
Phrasal Semantic Analysis
Phrasal Semantic Errors
Semantic Expressions + Phrasal Context
Sentential Semantic Analysis
Sentential Semantic Errors
Intermediate Representation + Sentential Context
Code Generation
IR Errors
Final Code

Figure 1: 4-Stage Compilation Process.

Resolution of pronouns is also performed in Stage 1, that binds the pronouns to the corresponding character. Moreover, it learns new words, such as 'happy' as discussed in the preceding section. Such words are not included in the original lexicon, and do not need to be real words. For example, the attribute could be 'xyz' as well. These newly learned words will be represented as variable attributes in the final code generation. We note also that the syntactic context, **SC**, generated for each sentence is also used to process the next sentence. This is because there might exist relations between consecutive sentences, with words such as "Otherwise" or if the sentence starts with a pronoun, etc.

The **SC** includes the tokens themselves and the statistics of the tokens such as the number of objects, the number and type of verbs, adjectives, colors, adverbs, numbers, etc. Some sentences may contain imprecise verbs, such as 'get' in "*When the fox gets the rabbit, ...*" Likewise, there might be usage of other verbs that do not mean their conventional semantics. These instances are also recorded in **SC**. Finally, conjunctives such as 'and' and 'or' are also recorded. The syntactic context, **SC**, along with the token stream, are passed to the next stage to generate the phrasal semantics.

Stage 2: Phrasal Semantic Analysis

In Stage 2, the generated tokens, **T**, and syntactic context, **SC**, from Stage 1 are used to generate semantic expressions, **SE**, and the corresponding phrasal context, **PC**. Consider a simple stream of tokens t_0, t_1, t_2: <obj, fox> <verb, chase> <obj, rabbit> for the text-phrase "*fox chases rabbit*". With frame semantics, the above token stream can be readily converted into a semantic expression of "verb(obj, obj)." Likewise, the phrase "*rabbit is chased by the fox*" maps to the semantic expression "verb(obj, obj)." However, the object identifiers should correctly correspond to the matching actor and actee.

Consider another more sophisticated phrase of "*the happy fox eats the rabbit that is not yellow*", the token stream, **T**, for this phrase produced from Stage 1 is t_i, … t_{i+4}: <adj, happy> <obj, fox> <verb, eat> <adj, not yellow> <obj, rabbit>. Note that the final modifier "*that is not yellow*" is converted to a token <adj, not yellow> in Stage 1 and placed as an adjective modifier *before* the final object. With this token stream, the semantic expression is "verb((adj, obj), (adj, obj))."

This stage also handles conjunctions. For phrases such as "*The foxes and tigers chase the rabbit*," two semantic expressions are generated internally, namely for the phrases "*the foxes chase the rabbit*" and "*the tigers chase the rabbit*".

Any error encountered in the process is also reported. Consider the phrase "*fox chases flees the rabbit*". The tokens would have been <obj, fox> <verb, chase> <verb, flee> <obj, rabbit>. In this case, consecutive verb-tokens are detected, and an error is reported for violating the rule of consecutive action verbs. This rule can be succinctly represented as $t_i \in A \rightarrow t_{i+1} \notin A$, where A is the set of action tokens. Essentially, this rule states that if the i^th^ token is an action, the next token must not be an action token.

Consider another erroneous example, "*fox chases happy*", the dangling adjective, 'happy', without any binding object is a violation and is reported to the user. Verb-tokens such as 'chase' require two objects around it. In this error, there was only one object, fox, which is insufficient to properly form the semantic expression.

Finally, when a conjunctive is about verbs, such as "*the foxes chase and eat the rabbits,*" the system will generate an error message noting the user that such sequences of actions should be split up into different sentences, and provide potential fixes such as "*The foxes chase the rabbits. When a fox catches a rabbit, it eats the rabbit.*" The above error violates the rule that prevents conjunction of action verbs "t_i and t_{i+1}", where both t_i , $t_{i+1} \in A$. The set of semantic rules is manually designed based on the valid phrases allowed in the system. Adding new rules to this set is straight-forward.

In addition to generating errors and semantic expressions in Stage 2, a corresponding phrasal context, **PC**, is also produced. The **PC** here refers to the set of semantic expressions (**SE**) discussed above, together with the number and types of semantic expressions, etc. For example, "fox sees rabbit" is a *relational* expression, while "fox chases rabbit" is an *action* expression. Other types of expressions include property expressions, such as "rabbit is happy", and variable expressions, such as "the size of the rabbit equals 3," etc. The **PC**, together with **SE** generated, are passed to Stage 3 to generate the sentential semantics.

Stage 3: Sentential Semantic Analysis

In Stage 3, the goal is to generate the intermediate representation (IR) for each sentence as well as the sentential context, **EC**. Consider the sentence, "*When a fox sees a rabbit, it chases the rabbit.*" After Stages 1 and 2, the semantic expressions are **SE**: se_0 = "if see(obj, obj)" and se_1 = "chase(obj, obj)." Each of the <obj> has a unique identifier to bind with the character in question. For this simple example, the IR for the entire sentence is "if see(obj, obj), then chase(obj, obj)." Consider another simple example with **SE** = "if property(obj, adj)" and "set_color(obj, col)". The resulting IR would be "if property(obj, adj), then set_color(obj, col)."

The types of errors in this stage include the following. Consider the sentence "*When a fox sees a rabbit, it sees the rabbit.*" Here, we have two relational semantic expressions involving predicates without any action expression. Thus, an error message will be produced for violating the missing actionable SE. Here, the violated rule is $\exists se_i \in$ action-SE for every sentence.

Re-writing of the phrases is also performed during this stage for some sentences. For example, if the sentence places the consequent before the antecedent, the system will internally re-write the sentence to preserve canonicity. Finally, resolution of conjunctives such as 'and' and 'or' are performed in this stage as well. Here, the conjunction is analyzed to determine if it is about two separate antecedents or consequents in the sentence. The set of rules for the sentential context is also manually derived, based on the sentences that combine various allowable phrases.

The sentential context, **EC**, for this stage includes the set of IR, together with the type of the IR, as well as the number of antecedents, consequents, complexity of the antecedent, etc. For example, the sentence "*When 35 rabbits are gone, …*" contains a counter 35, along with the IR for the sentence that is given to the subsequent code generation stage.

Stage 4: Code Generation

Finally, with the IR and sentential context, **EC**, Stage 4 generates the output game code based on the **EC**. If there are no errors in any of the previous stages, the IR from Stage 3 would be readily translated to the game code. On the other hand, if there are errors, the context vectors are used to help fill the gap(s) when generating the code. For example, in the first stage, if the number of object tokens is significantly greater than the number of verb tokens (or vice versa), we analyze the token stream further to generate both the code and any additional error message, if appropriate. For instance, in the consequent phrase "*it chases it*", if there are two characters in the antecedent, the system will fill in the two pronouns according to the characters in the antecedent.

We had briefly touched on variables earlier. In addition to Boolean variables such as 'happy', the system also handles non-Boolean variables, such as 'size' in "*When the size of the fox is less than 5, …*" Here, 'size' is a built-in variable available for every object. The user can also define new variables, such as "*When the num_eaten of the rabbit is equal to 5, …*" The variables can also be used in modifier clauses as well. The following sentence is one such example: "*When a rabbit whose size is less than 10 sees a fox, it turns red.*" Here, the phrase "*whose size is less than 10*" modifies the rabbit object in the antecedent.

4 Results

The current 4-stage process helps in both the translation of the input text and error-reporting. With this platform (Game Changineer), we are able to produce a wide range of error messages and offer possible fixes to the error. For example, the sentence "*When a rabbit sees a fox, it chases it*" is ambiguous as discussed earlier. The system generates an ambiguity error (noting the two pronouns). Nevertheless, in the presence of such an error, the system will still produce an approximate understanding so that a final game code can still be produced (so that the user can test the game).

Consider another erroneous sentence: "*When a fox sees a rabbit, it chases.*" The verb in the consequent is an action verb (chase), and it is missing a target object. Thus, an error is reported. In addition, the engine tries to remedy the semantic expression by inserting the most suitable missing object from the former semantic expression(s). In this case, it would be the fox chasing the rabbit. This temporary filling of the missing object allows the code generation to complete its generation of the game code. Nevertheless, the above error message is still provided to allow the user to fix the error.

Consider a third error example: "*When the rabbit shoots the fox, the fox runs away.*" This sentence may seem correct at first glance, but it is actually ambiguous on the word 'shoot'. Recall that the sentence may be sloppily written by a young user, and as with any NL, the manner in which a verb is used may be imprecise. In this case, there are two possible interpretations of the antecedent clause: (1) '*when the rabbit fires a bullet at the fox*' or (2) '*when a bullet touches the fox.*' With the first interpretation, we know that not every bullet fired will hit the fox. In fact, many bullets might actually miss the fox. This corresponding error violates the imprecise antecedent verb.

We believe that a good NL-based programming platform should provide a helpful debugging infrastructure to give feedback and guidance to the user on possible misinterpretations. With the error-reporting framework, the system has been piloted at several outreach events to middle school students in the 2020-2021 school year. To the best of our knowledge, no other publicly available system exists that allows users to write video games in English, generates feedback and suggestions on how to fix syntactic and logical errors in the natural language sentences. Because there is no public dataset available, the results are tabulated on anonymized input sentences written by 434 middle school students during the month of March, 2021. Each student created a number of games during the month, and each game may require multiple iterations of debugging. Among the 47,907 errors collected, the 10 most-frequent-occurring errors are reported in Table 1. Both the number of occurrences and type of error are shown.

Table 1: Most frequent-occurring errors

# occur	Error type
742	Spelling error
604	Imprecise verb (such as 'get')
382	Unclear / unsupported phrases
296	Move without direction
294	Incomplete sentence
247	Missing a valid character
245	Ambiguous antecedent
224	Missing a valid verb
223	Imprecise word
178	Logical error on sequencing events

Based on Table 1, it is not surprising that spelling error ranked highest. It is worth noting that the system may generate multiple errors for a given erroneous sentence. For example, a spelling error may also result in a "missing a valid character" or "unclear / unsupported phrase" error. The second most frequent error was the use of imprecise verbs. These occur with phrases such as "*fox gets the rabbit*" or "*the bird is hit*". These phrases can have multiple interpretations, including "touch", "eliminates", or "is shot". Next, unclear / unsupported phrases include those facetious phrases such as "the fox farts". Next, an example of "move without direction" is the phrase "*the rabbit moves at 2 pixels per frame*". This can be interpreted as either "*wanders around at 2 pixels per frame*" or "*the speed of the rabbit is 2*." Without clarity, the system chooses the latter.

An example of an incomplete sentence is "*When the rabbit sees a carrot, chase.*" Here the objects in the consequent are missing and need to be filled. Finally, the logical error on sequencing events is an interesting type of error. For example, a sentence "*When the rabbit dies, the game is over*" is correct in itself, but will result in such an error if there was no earlier description on how the rabbit can die before this sentence. Screen shots of two games created are illustrated in Figure 2. Many more games are available on the website.

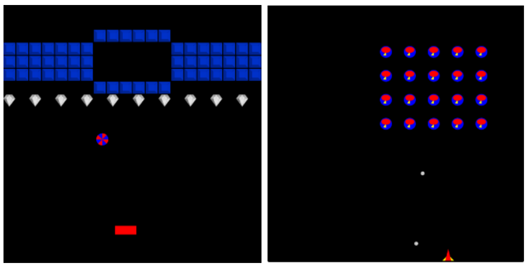
Figure 2: Breakout and Space Invaders Type Games

5 Conclusions

We have presented a 4-stage process to generate error messages for English sentences that could not be processed. At each stage, a context vector is constructed and propagated to the next stage. Analysis of the context vectors plays a critical role in both the generation of game code and any error messages that pinpoint imprecise, incomplete, and/or incorrect expressions. These error messages help guide the user to correct their errors. Results from games created by Middle-school students show the potential of such a framework to help them bring their designs to completion.

References

Game Changineer website: https://gc.ece.vt.edu

Hsiao, M. S. (2018). Automated program synthesis from object-oriented natural language for computer games. Proc. Int. Workshop on Controlled Natural Language.

Zhan. Y., & Hsiao, M. S. (2019). Multi-label classification on natural language sentences for video game design. Proc. IEEE Int. Conf. on Humanized Computing and Communication.