Incorporating Timing into the Formal Glossa Language for Mapping NL Sentences to an SPN State Machine for Agent Action Patient Association

Alena McCain

Abstract— Although significant improvements have been made in Natural Language Processing and Natural Language Understanding, there is still much ambiguity in the methodologies proposed to improve artificial intelligence in these fields. For example, many methodologies still consider the timing of actions in the kernels to be arbitrary and uncertain when the intelligent agent attempts to analyze them. However, determining the relative timing of sentences and kernels close to each other is a crucial aspect of human understanding of natural language. Therefore, it should be present in all such algorithms. Unfortunately, however, that is not the case. This research shows that minor improvements can be made to one such method (the formal language Glossa) to allow the intelligent agent to know the timing of certain SPNs near each other by looking for timing separator words in sentences like humans do. In addition, this methodology allows for a better understanding of the algorithmic process from the addition of user input when uncertainty is involved and sample times for previously random SPNs.

Keywords—Kernel, Glossa, SPN, Agent, Action, Patient, State Machine

# Introduction

While the fields of Natural Language Processing (NLP) and Natural Language Understanding (NLU) have been studied by computer scientists for many decades, there is still much research to explore because of the inherent uncertainty and ambiguity in natural language.

The research presented in this paper provides an extension of the Stochastic Petri Nets (SPNs), state machine methodology presented by Psarologou and Bourbakis in 2016 [1]. Even though this paper does not produce a complete replication of the work, enough of the original work was preserved to provide a proof of concept for this paper’s proposed additions to the work. Like the original work, this paper uses kernel extraction (Agent Action Patient or AVP) from sentences and kernel conversion into the formal language Glossa. Finally, the Glossa is used to create SPNs for the state machine.

However, there is still much uncertainty in the previously proposed methodology due to the ambiguity in ordering natural language sentences and kernels relative to each other. Therefore, this research aims to provide a methodology for more accurately determining the timing of natural language sentences near one another and the timing of kernels within a natural language sentence by using words in the sentence that denote an order. The formal language Glossa introduced in the previous paper is also updated. In addition, this methodology also aims to reduce uncertainty by allowing the user to set probabilities of each kernel or AVP occurring in the state machine based on prior knowledge when there are many possible kernel or AVP combinations.

This paper is organized as follows. Section 2 briefly explains how the ideas and concepts from the previous work were implemented in this research. In addition, the omitted functionality from the previous work is presented. Section 3 describes the additions and improvements made in this methodology from the previous work. Section 4 compares the SPNs of the previous and current work and gives examples of the SPNs created by the additions in this methodology. Finally, section 5 presents the conclusions of this research and potential future work.

# Application of Previous Implementation in This Work

## General Implementation

The program from the previous work was implemented using Java and used the Stanford Parser. Python 3 and the Python text parsing library Spacy were used in this work’s program. The change in the programming language is because Python is much more widely used for artificial intelligence applications than Java. In addition, Spacy’s ability to distinguish words based on multiple factors, part of speech, tag, and dependency (a word’s relation to another word), allowed for greater flexibility in creating sentence parsing rules. Similarly, since a sentence is ultimately a list of words, Python’s list comprehension abilities significantly improved ease of programming and execution time.

## Kernel Generation

This implementation’s kernel generation produces the same results as the kernels produced in the previous work. However, a few modifications to the algorithm presented in the previous paper were made. Figure 1 shows the algorithm used in this work.

First, all nouns and verbs are found within a sentence. Then, the algorithm finds consecutive verbs to determine where each verb phrase begins in the sentence. These are considered the end of a clause. Next, the algorithm looks at all the words between each verb phrase to determine where the patients for the first verb phrase end and the agents for the following verb phrase begin. That process is done by looking at the placement of commas and conjunctions relative to all the applicable nouns. If there are multiple options for separating the agents and patients, the greediest separation was selected for simplicity. In this context, greedy means that the separation with the most patients for the first verb phrase is chosen. Therefore, this work may only produce a subset of the kernel options produced in the previous work if they considered multiple agent patient separations between kernels. The first agent between the verb phrases is considered the start of a clause. Also, the first word in the sentence, excluding the prepositional phrase, is considered the start of the first clause.

Next, the last comma or conjunction between the end of one clause and the start of the following clause is found for every pair of clause indices. These indices are the kernel separators that determine where each kernel starts and ends. Then the following is executed for each pair of clause separators. First, the verbs and their connections are found. These are the actions. After that, the nouns after the first clause separator and before the verb phrase and their connections are found. These are the agents of the verb phrase. Lastly, the nouns and their connections before the first clause separator and after the verb phrase are found. These are the patients of the current verb phrase.

All agents, actions, and patients are found using the constraints outlined in section 2 of the previous work. However, since the table of prepositional verbs was not provided in the previous work, constraint seven was omitted from this work.

In addition, if there are multiple verb phrases in a kernel, and at least one of them contains multiple adverbs, it is assumed that all the adverbs in each verb phrase occur in the same position; before or after the main verb. This implementation was chosen because consecutive adverbs do not have to be separated by a conjunction, causing potential ambiguity if other verb phrases contain different adverb positions relative to the main verb. Similarly, the main verbs that an adverb describes are ambiguous if more than one main verb is in the sentence. For example, the sentence “Travis slowly drank, dropped, and spilled the drink.” It is unclear whether slowly describes only the first main verb or all of the main verbs. Therefore, main verbs are only considered to be described by adverbs connected directly to them. Likewise, it is assumed that only one conjunction is between adverbs. Allowing for multiple conjunctions would dramatically increase computational complexity because the adverbs would have to be distributed to the main verb in the correct combinations.

In addition, the program doesn’t accept adverbs separated by “or” conjunctions because the conjunction grouping could be incorrectly interpreted when creating Glossa. For example, the sentence “Travis slowly or quickly ate or ate and played with the food” contains the action phrases and conjunctions; slowly ate, or, quickly ate, or, ate, and, played with. The “or” conjunction that was not in the original sentence could result in an error because of the recursion in the Glossa creation algorithm (See subsection C). Adding functionality to include “or” conjunctions would also increase computational complexity. This issue is not present with “and” conjunctions because those conjunctions are removed so they can be combined with the main verb to form the action. For example, “slowly and steadily ate” would be converted to the action slowly\_steadily\_ate in the SPN. Fortunately, none of these omissions affect the results presented in this paper.

Even though the generated kernels from this work resemble the ones found in the previous work, it should be noted that, unlike the previous work, the adjectives, numbers, and other descriptive information are not saved and stored in conjunction with the kernel. The information was omitted because it is not pertinent to the central concept described in this paper. However, the sentence is stored with the kernel, and adding this additional information later would be reasonably trivial because of how the program was organized.

## Mapping kernels to Glossa

While the Glossa language presented in this work follows the previous work, this approach follows a much more deterministic pattern. There is only one possible Glossa conversion for each sentence. In previous work, generating multiple Glossa combinations was quite common due to the ambiguity of natural language when using multiple conjunctions in a sentence (as mentioned briefly in subsection B).

However, this work uses a recursive approach to determine how each set of agents, actions, patients, or kernels is separated by conjunctions when converted to Glossa. Therefore, the central conjunction combines the largest number of agents, actions, patients, or kernels. The sentences below show several examples of this. The algorithm essentially separates the agents, actions, patients, or kernels into two halves based on the center conjunction in the part of the sentence currently being analyzed. The algorithm continually finds the middle conjunction of each half until each half only contains one agent, action, patient, kernel, or conjunction. Then the previous halves are grouped accordingly as the recursive calls are exited. Much like in subsection B, the deterministic nature of this algorithm is used for simplicity. Therefore, it doesn’t affect the results presented in this paper.

[(Lilly and Jake), or (Mark and Travis)] played chess.

Mark teased [(Lilly, Jake, and John), or Sarah]

[([Jake or Lilly], and [Mark, Sarah, or Travis]) played chess] and [(John and Tom) played checkers].

## The State Machine and SPNs

The state machine and SPNs closely mimic the implementation in the previous paper, even though the SPNs in this paper are represented in a text-based manner instead of a graphical manner to demonstrate the timing aspect of NLU better.

Figure 2 shows an example of an SPN produced from this program. As one can see, this output provides much more contextual information compared to the graphical output presented in the previous work. Most of these differences are discussed in section 3. First, however, it is essential to note that the graphical output implicitly shows the movement from the pre-action state to the post-action state. In contrast, the text-based output explicitly displays every state of the SPN.

Much like in the previous work, the state machine uses tokens to determine the AVP combinations to execute. If the algorithm must choose a subset of agents, actions, patients, or kernels to execute, the program accepts user input to determine the probability of certain AVP combinations occurring. Examples of this are shown below in Figures 3 and 4. Section 3 provides a deeper explanation of how the state machine runs because the implementation and execution of the state machine from the previous paper is not well documented from an algorithmic standpoint. Therefore, comments cannot be made regarding this implementation versus the previous implementation. In addition, much of this work state machine is related to the new work presented in this paper

One significant omission from this work is that the SPNs are not combined, and Anaphora Resolution was not used. They could potentially be implemented later but were not relevant to the main point of this paper. Therefore, they were omitted for simplicity. Specifically, AR would have to be investigated more to determine how it works with Python since the previous paper references the Stanford NER tool for implementation.

# Additions to the Previous Work

## Timing Separator Operator in the Glossa Language

The main addition of this work from the previous work is the addition of the timing separator operator (#!) in the Glossa. The operator distinguishes a coma as part of conjunction between kernels and a comma separating two kernels with a timing preposition. Examples of this distinction and how the previous work would have misinterpreted when kernels should execute are shown in Figure 5. Kernels of the same color are executed at the same time.

This distinction and operator are essential because it allows the algorithm to determine the timing of kernels or sentences that are next to each other. The examples in Figure 6 show the relative timing values given to kernels in sentences that contain timing words. These examples always assign the first kernel a zero “order.” The following kernels are set accordingly. The random time assigned to the kernels preserves their order when the state machine is executed.

There are a few steps to determine where the timing separator operator should be placed in the sentence Glossa. First, it needs to be determined if there is a timing word in the first, last, or intermediate kernel. If the last case is true, the timing separator operator is inserted immediately before the timing word. On the other hand, if there is a timing word at the beginning of the first or end of the last kernel, the algorithm recursively iterates through all the conjunctions separating the kernels until two kernels are found separated by a comma without a corresponding conjunction. Then, the timing separator operator is inserted at the position of the comma. If no two kernels fit this criterion, there is no need for the timing separator operator, which Implies that the ordering is between two sentences instead of kernels within a sentence. Examples of this process are shown in Figure 7. The relevant timing word is in yellow, conjunctions are in blue, and the position of the timing separator is in green.

In the algorithm, once a kernel that contains a timing word is found, the order number of subsequent or previous kernels or sentences is updated to reflect the order induced by the timing word. The algorithm for this process is shown in Figure 8. It should be noted that the phrase order number is intentional. At this stage, the algorithm is not assigning random times to the sentence and kernels but rather determining the order of the sentence Glossa that the state machine later converts into SPNs. Each SPN state is assigned a time when the SPN is created.

The following timing root words and phrases are currently implemented in this algorithm; "while", "simultaneous", “at the same time", "in the meantime", "during that time", “before", "earlier", "previous", "prior", “after", "later", and "next". Even though this is by no means an exhaustive list, it still captures many of the timing words used in everyday English. It should also be noted that these words are also considered roots. Therefore, a word like after matches itself and afterward. Similarly, words like simultaneous and previous can match simultaneously and previously.

A few words and phrases must be noted because of their inherent ambiguity. These are while, simultaneous, at the same time, in the meantime, and during that time. Given the current usage of these words in everyday English, it is impossible to determine if the AVP combinations are being completed simultaneously, meaning they have the same start and end time, or they start at the same time and could end at different times. In this algorithm, the latter implementation was chosen because it is rare in the understanding of everyday language that if someone says they’re going to do something while someone else is doing something, they don’t necessarily intend to finish their actions simultaneously. For example, my mom might say she’s going to the grocery store while my dad is at work. However, this doesn’t mean that she knows exactly when my dad intends to finish working or when she intends to finish shopping. It should be noted that the same assumption is also used when looking at the “and” conjunction between kernels because similar ambiguity applies.

## The SPN

As mentioned in section 2, the SPNs produced in this algorithm are text-based instead of graphical. Using text allows for printing the timing of each state in the SPN. The SPNs have three states in this implementation: pre-action, action, and post-action. The pre-action state is essentially a waiting state before the relevant agents perform an action on the relevant patients. In this state, if there are multiple options for agents or patients (AVP combinations), all of them are considered busy. Once the selected agents and patients start performing the action (the action state), the agents and patients in the other AVP combinations are free to perform other actions. The post-action state occurs after the agent has completed the action and the patient has received the action. At that point, none of the relevant agents or patients are considered busy. An example of each state’s busy agents and patients is shown in Table I.

## Executing the State Machine with Timing

In this implementation, the state machine starts by ordering the sentence Glossa in ascending order based on their lowest kernel Glossa order numbers (see subsection B). Then the sentence Glossa with the lowest order number is selected. The examples in Figure 9 show the difference between sentence Glossa and kernel Glossa for clarity. Before analyzing a single sentence Glossa, the algorithm checks the list to see if the following sentence has the same order number as the previous one. If so, the current and the following sentence Glossa are combined using the #! operator (to denote that they execute simultaneously). An example of this scenario would be John teased Kate. At the same time, Sarah teased Mike.

Then all the kernel Glossa within the sentence Glossa are sorted in ascending order by their order numbers. Then the algorithm loops through all the order numbers within the sentence Glossa. In the loop, all possible combinations of kernel Glossa with the current order number are determined based on the adjoining kernel operators. All possible AVP combinations for a specific Glossa kernel combination are found simultaneously. Then the algorithm presents all the possible kernel Glossa combinations and selects a random combination based on user input, if applicable. After selecting one kernel Glossa combination, the algorithm presents all the possible AVP combinations of the kernel Glossa combination. Like the kernel Glossa combination selection, the algorithm selects a random combination based on user input, if applicable. The user input is the probability of each kernel or AVP combination being selected based on prior human knowledge. The input allows the user to play with kernels and AVP combination probabilities and see how the output is affected when different values are input. A demonstration of these steps is shown in Figures 2 and 3.

Lastly, the algorithm assigns all the kernel Glossa state transition times (pre-action, action, post-action) based on whether the order number of the kernel Glossa is consecutive or random compared to the previous order number. If it is consecutive, the pre-action transition time is selected based on the previous kernel Glossa’s post-action transition time. Furthermore, the pre-action time must be sometime after the post-action time. On the other hand, if the order is random, the pre-action transition time can happen anytime between the start and end of the simulation.

It is important to note that only one AVP combination is selected if multiple AVP combinations are in one kernel Glossa. If this is true, the agents and patients in each combination are used to determine the pre-action transition time. Therefore, all the agents and patients in every combination are considered busy until the pre-action state is reached. Then all the agents and patients in the AVP combinations that weren’t selected are now available to complete other actions, as briefly explained in subsection B. In contrast, the action and post-action transition times are selected randomly between the pre-action transition and the next time that any agents and patients in only the chosen AVP combination start completing a different action. Finally, the token specifying the transition times, agents, actions, and patients of the kernel Glossa is added to the state machine to execute the states at their assigned transition times. The algorithm for initializing the state machine is shown in Figure 10.

# Results

## Original Results

Figures 11 through 13 show the resulting SPNs created by this program from multiple sentences when there is uncertainty in the SPN timing. It is important to note that the pre-action time of each AVP combination is arbitrary unless it is immediately before, after, or at the same time as another AVP combination’s time based on timing word placement. This program merely provides one sample timing for all the sentences, which can be seen when comparing the runs in the figures. It should be noted that the only difference between the resulting SPNs in each run is the timing of the random kernels (because none of these sentences include an or conjunction).

Figures 14 through 17 show the resulting SPNs created by this program from multiple sentences when there is uncertainty in the kernel combinations, the AVP combinations, or both. As one can see, this type of input provides for many possible interpretations of the sentences due to a large amount of uncertainty because of the random kernel selections, AVP selections, and timing.

## Results Compared to Previous Work.

Figures 18 and 19 show a comparison of the output of this work and the previous work, based on the following paragraph:

“I saw Tom with the telescope. The telescope sat on the table. The table and telescope was seen by all of us. We had a great time at the party. It was Tom’s birthday. Tom received several presents from all of us. The telescope was Tom’s favorite present. The telescope appeared expensive. I looked through the telescope. A great time was had by all.”

As one can see, both programs produce consistent output (except for “The telescope sat on the table” because of the omission of one of the kernel action constraints, as discussed in section 2). However, the previous work does a better job of presenting the ambiguous timing of the kernels because it doesn’t provide a sample timing like the program presented in this research. The importance of ambiguous timing is especially prevalent when looking at the sentences “The telescope appeared expensive” and “It was Tom’s birthday.” When interpreted by a person, these sentences and their SPNs should be able to run simultaneously with other SPNs, even if they have the same agents or patients, because the sentences merely provide descriptions of nouns as opposed to an agent performing an action to a patient. One could argue that the second sentence does have a finite time because it describes the time of a day. Similarly, the first sentence arguably has no end time in this context. Both discrepancies would imply that the sample timing mechanism for this algorithm has a minor flaw depending on the context in which it’s used.

Even though providing sample timing is not the most useful for simple sentences like in the paragraph presented above, the time value provides the much-needed context concerning the potential for multiple SPNs to execute in a particular order. In this regard, neither implementation is better or worse than the other. It mostly comes down to a personal preference of whether or not the user would like the timing of the SPNs to be arbitrary or to be able to look at multiple examples with different time intervals.

# Conclusions, Obstacles, and Future Work

## Obstacles

Many obstacles in this research were incurred due to omissions of some methodologies from the previous work. It was not obvious how to implement them in Python. The main omissions are using a parse tree to determine all possible AVP and kernel combinations, combining SPNs, and Anaphora Resolution. While the point of this research could still be obtained with the omissions, it would have been more beneficial to implement them. For instance, more sentence types could have been parsed, and more kernel and AVP combinations could have been explored.

In addition, while the use of the Spacy library provided much-appreciated flexibility when it came to parsing text for kernels, it also made the process much more manual and labor-intensive than using the Stanford parser in Java. While this is not a huge obstacle, it is still something to consider when adding constraints to this parsing algorithm in the future.

## Conclusions

Even though previous work in the field of NLU led to the formal language Glossa, it missed one key aspect of NLU: more concrete timing of kernels by scanning the text for timing words and phrases. This work explored the idea and provided the basic algorithm for finding timing words in sentences and using rule-based artificial intelligence to determine the ordering of kernels and sentences near each other.

The text-based output of this work provides a better means of showing the timing relation between SPNs compared to the previous work. The only drawback is that by making the SPN timing so concrete, one loses the ability to generalize the order of the sentences that don’t have an apparent order. In addition to sentences that contain standard agents, actions, and patients, sentences that are merely descriptive or have a specific start and end time based on context are also affected by the algorithm’s lack of generalization. On the other hand, this method streamlines the need for the user to determine the potential execution order of SPNs. In addition, this algorithm allows the user to query by providing different probabilities of kernel and AVP combinations.

While the algorithm presented in this research omitted some of the more robust and thorough tactics from the previous work, it still improves the overall algorithm for NLU and the formal language Glossa. Furthermore, the practices omitted in this work could easily be implemented in the future and do not affect the results of what this research set out to explore. Fortunately, writing this program in Python and using the Spacy library allows for much more flexible updates to this program in the future.

While the invention of Glossa was revolutionary, the previous work was much more general by just exploring how to get from a piece of text to SPNs. However, the improvements in this work allow the user to explore the intelligent process behind NLU and help them create improvements in the field.

## Future Work

It would be beneficial to include a parse tree, Anaphora Resolution, and combined SPNs in this program in the future. Therefore, the algorithm can have all the current improvements and valuable tools presented in previous implementations. In addition, allowing for more sentence types would be helpful. For example, the program could allow adverbs separated by multiple conjunctions (including “or” conjunctions in verb phrases or to describe multiple main verbs (if applicable).

It may also be beneficial to determine a method to show the ordering of nearby kernels and sentences without setting random timing for kernels and sentences that are not ordered. Then, the fundamental concepts of this paper would still be shown while retaining the underlying ambiguity in the timing of most SPNs, as presented in the previous work.

##### References

1. Psarologou, Adamantia and Nikolaos G. Bourbakis. “Glossa - A Formal Language as a Mapping Mechanism of NL Sentences into SPN State Machine for Actions/Events Association.” *Int. J. Artif. Intell. Tools* 26 (2017): 1750012:1-1750012:31.