Extracting Temporal PDDL Code from Natural Language using Large Language Models

Alexander Beamish, 19akb3@queensu.ca, Queen's University

Abstract

The task of converting natural language descriptions of actions to Planning Domain Definition Language (PDDL) code has interested many researchers, and many techniques have been applied to this task in the past. One aspect of this task that has not been focused on is the extraction of condition and effect statements for durative actions. These statements have a unique syntax in PDDL that allows them to express the point in time at which the condition or effect applies, relative to the action. The goal of the present research was to develop a model that could successfully generate these statements based on natural language descriptions of the statements. For the purposes of this research, a novel dataset was created and used to generate text-based prompts for each piece of natural language input. The prompts were fed into a large language model (LLM), and the output of the LLM was processed in order to obtain PDDL code corresponding to the condition and effect statements described in the input. Appropriate PDDL code was successfully generated for relatively simple descriptions of conditions and effects, and more complicated input could likely be handled if the dataset was expanded. The results of this research demonstrate the great potential of using LLMs for automatic code generation from natural language.

1 Introduction

The task of accurately modeling problem scenarios in Planning Domain Definition Language (PDDL) domain files is often difficult and time consuming. Moreover, the subject matter experts who understand the problem well enough to do the modeling are often not versed in PDDL syntax [1]. Given the difficulty of manually encoding domain files, it is not surprising that researchers have attempted to automate the process of generating the PDDL code for these domain files from natural language (NL) [6]. Within this area of research, many different approaches have been applied for many different purposes. As discussed in Section 6, some researchers have attempted to induce full PDDL domains from NL. The present research takes a more focused approach in that it does not aim to generate complete PDDL domains, but only to generate specific components of domains. Additionally, the present research focuses on temporal PDDL and durative actions. The specific focus of the present research is to generate condition and effect statements for durative actions from NL descriptions of these statements.

The approach taken in the present research involves storing a database of NL text that describes PDDL code, and using this database to predict lines of PDDL code for unseen NL input, with the help of a large language model (LLM) such as GPT3. This general strategy is straightforward, but the implementation of a system that accomplishes this could be done in many different ways, using many different techniques. The goal of this research is to test the viability of this general approach by creating a relatively simple system that converts NL descriptions to temporal PDDL condition and effect statements. The creation and distribution of the novel dataset used by this system also constitutes a significant contribution of the present research to the planning community. If the general approach taken in this research shows promise, then future work can be done to expand and improve both the dataset and the system.

2 Preliminaries

PDDL

Automated planning is an active area of research within the field of artificial intelligence. PDDL is a language specifically designed for representing planning problems. A complete representation of a planning problem in PDDL consists of a domain file and a problem file. The domain file contains the predicates and actions, while the problem file contains the objects, the initial state and the goal state. The domain file defines the nature of the world that the problem is set in, while the problem file provides details about the specific problem that is to be solved [4]. PDDL domains have traditionally been hand-coded by individuals with knowledge of PDDL syntax alongside subject matter experts, and the process of creating these domains is often time-consuming [5].

Classical and Temporal Planning

Much planning research has focused on classical planning, which is characterized by a set of assumptions that are outlined in [6]. Among these assumptions is the assumption that actions have no duration. Given that this assumption

is unrealistic for many actions in real-world scenarios, it is not surprising that researchers have attempted to go beyond the classical planning framework by allowing actions to have duration. The term "temporal planning" is used to describe planning where the actions have duration. Support for durative actions was introduced in PDDL 2.1 [1]. The syntax for a durative action is similar to that of a normal PDDL action, but there is an extra statement that expresses the action's duration, and there are also differences in the condition and effect statements for the action. Examples of durative actions can be found in Figure 2, located in Section 5.

Large Language Models

A large language model (LLM) is a machine learning model that can recognize and generate language based on an extensive text-based dataset. Generative Pre-trained Transformer 3 (GPT3) is an example of a LLM that takes a prompt as input and outputs text that naturally follows from the prompt, according to the model. GPT3 is able to detect simple patterns that are present in the prompt, and use its large database to produce output that mirrors the prompt in form, and that naturally follows from the final line of the prompt. An example of a prompt, and of the resulting output of a LLM, are shown in Figure 1, located in Section 5. The present research makes extensive use of the capabilities of LLMs to generate PDDL code.

Natural Language Processing Techniques

Several standard natural language processing (NLP) techniques are employed in this research. As is common practice for most NLP tasks, text is converted to a vector representation. This allows for the simple comparison of sentences using a cosine similarity metric. The comparison of sentences is integral to the process of generating appropriate prompts for a LLM, so many cosine similarities are computed every time the model is run. The cosine similarity is computed for vectors A and B as:

$$cosine\ similarity = \frac{A \cdot B}{||A|| \cdot ||B||}$$

3 Approach

Overview

The general approach taken was to create a novel dataset consisting of annotations of durative actions in PDDL, and use this dataset to create individualized LLM prompts for each piece of natural language input. Ideally, these LLM prompts would then result in output from a LLM that includes the PDDL code described in the input. In order to filter out incorrect output, and obtain just the desired PDDL code, some simple processing was performed on the output of the LLM. Although this general approach is simple and intuitive, the creation of a system of this nature required the integration of several components, each of which could be implemented in different ways. The details of the implementation of this system will be explained in this section.

Database

After searching for data that could be used to generate appropriate LLM prompts, it was determined that it would be necessary to create a novel dataset by annotating existing temporal PDDL code. The resulting novel dataset would not only be useful for the purposes of the present research, but could be made freely available so that other researchers in the planning community could make use of it as well. More details about this novel dataset can be found in Section 5. In the present section, the role of this data within the system for generating PDDL code from natural language shall be examined.

Parameter and Predicate Selection

The model created takes as input one or more natural language sentences describing some or all of a durative action, and generates an LLM prompt for each sentence in the input. Before processing the input, an interactive feature of the model allows a user to select parameters and predicates from a list of suggestions. The mechanism that selects words to suggest as parameters and predicates is very primitive, and relies primarily on part of speech (POS) tagging. A more advanced mechanism could be implemented in the future, but was not implemented in the present model because extracting parameters and predicates is not the primary goal of the present research. After the parameters and predicates have been selected by a user, the process of generating a LLM prompt for each sentence in the input begins.

LLM Prompt Generation

The prompt generation process involves several steps. Firstly, the input is separated into a list of sentences, and all parameters and predicates in each sentence are replaced by the strings 'param' and 'pred'. Then, a vector representation of each sentence is computed. After the input sentences have been modified and converted to vectors, each input sentence is compared to each sentence in the database of annotations. The embeddings for each sentence in the database were computed and saved prior to the running of the model, and the parameters and predicates in these sentences were modified in the same way as the those in the input sentence were, prior to computing the embeddings. Since each sentence in the input and each sentence in the database has been modified in this way, the comparisons should reveal the sentences in the database which are most similar in form to the input sentence, and differences in the names of specific parameters and predicates should be ignored. The comparison is done by calculating the cosine similarity between the vectors of the two sentences being compared. Once a sentence in the input has been compared to every sentence in the database, the 10 most similar sentences in the database are used to generate the

individualized prompt for that input sentence. The number 10 is arbitrary, but yielded good results in terms of the output of the LLM.

Each prompt requires not only natural language, but PDDL code as well. If the specific line of PDDL code that is described by individual sentences in the database is known, then the creation of LLM prompts is straightforward. With this information, it is possible to place the corresponding line of PDDL code directly after each piece of natural language text that is used in the prompt. By placing lines of PDDL code directly after natural language sentences, and then appending the natural language input to the end, it is possible to create a sensible prompt for a LLM. An important step, then, is to establish some kind of link between natural language sentences in the database and specific lines of PDDL code.

Since the PDDL code that corresponds to each annotation in the dataset is stored along with the annotations, it is possible to roughly estimate which line of PDDL code corresponds to which sentence in a given annotation using the cosine similarity. To do this, the cosine similarity was calculated between modified lines of PDDL code and sentences in the dataset. The modification of the PDDL code was simply a removal of parenthesis and question marks. While this strategy for linking sentences to lines of PDDL code is not perfect, it is practical because it is easy to compute, and it is reasonably accurate.

It would have also been possible to either have annotators annotate single lines of PDDL code rather than complete actions, or to manually link parts of annotations to lines of PDDL code, but this was not done for several reasons. Complete actions were annotated because annotating single lines of PDDL code would have resulted in more formulaic annotations that would not have covered the full range of possible ways of describing the actions. Moreover, if the dataset was expanded in the future, it might not be possible to find such neatly organized annotations, and manually linking sentences to lines of PDDL code would also be impractical with more data. Therefore, it was deemed more practical to test whether an automatic method of linking natural language sentences to lines of PDDL code could function adequately. With the cosine similarity metric, it is not only possible to link single sentences in an annotation to single lines of PDDL code, but it is possible to obtain percentage values representing the degree of similarity of each sentence in an annotation to each line in the corresponding PDDL code. This is especially beneficial in cases where a single sentence describes multiple lines of PDDL code, or when multiple sentences describe a single line of PDDL code.

The entire process of prompt generation for a given input can be summarized in 3 steps. The first step is to find the 10 sentences in the database that are most similar to the input sentence. The second step is to use previously calculated similarity measures between these sentences and lines of PDDL code in order to determine the specific lines of code that each similar sentence describes, and then place the most similar lines of PDDL code directly after the corresponding sentences in the prompt. The final step is to append the natural language input to the end of the prompt. The input was repeated 3 times at the end of each prompt in order to make the amount of text in the final line more similar to the amount in previous lines of the prompt. An example of a prompt that was generated using this process is shown in Figure 1.

Processing of LLM Output

Although the form of the prompt is designed to make the LLM output PDDL code, it does not guarantee that the output is sensible. Ideally, the output of the LLM would be precisely the PDDL code described in the input sentence. However, the LLM is not always so precise, and it sometimes outputs either syntactically incorrect code or code that is not described in the input sentence. In order to compensate for the unpredictability of the output, it was deemed necessary to filter out the parts of the LLM output that are obviously not correct.

The filtering process was done in 3 steps. The first step was to convert the output of the LLM into a list of code segments, if possible. In most cases, the output of the LLM had the same form as a typical condition or effect statement in PDDL, meaning that it began with the substring ‘(and’, followed by several substrings representing individual conditions or effects. If the output is in this form, then it is possible to eliminate the substring ‘(and’, and to store each condition or effect in a single list. If the output is not in this form, then it is not worth processing at all because it is almost certainly nonsense. Assuming that the output can be broken up into a list, the second step is to check each code segment in the list to see whether it contains a predicate that is not present in the input sentence. If it does, then the code segment should be eliminated from the list. This is a simple way to check whether the code segment could possibly correspond to a condition or effect described in the input sentence. The third step is to check the syntax of each code segment. If the code segment does not have correct syntax, then it should be eliminated from the list. An exception to this rule was made if the code segment had an incorrect number of right parentheses, as this occurred frequently and was simple to correct for. It is important to note that these steps do not ensure that the output of the model is correct, but they do filter out parts of the output that are obviously not correct.

Duration Extraction and User Interface Design

Although the primary focus of the present research was the extraction of PDDL condition and effect statements from natural language, duration statements were also extracted using simpler techniques. It was observed that all of the annotations used words from a small set of words that can describe a duration. These words include ”takes”, ”duration”, and ”lasts”. Checking whether these words were present in an input sentence, and then extracting digits from this sentence proved to be a sufficient method for extracting the durations, at least for the purposes of the present research.

The goal of the duration extraction was primarily to allow for the complete code for a durative action to be displayed after the model is run. The parameters and predicates are discovered through user interaction, the condition and effect statements are generated by prompting a LLM, and the basic skeleton for a durative action is easy to create, so finding the duration using a crude method allowed the complete code to be displayed to the user. For the purposes of testing and demonstrating the model, a simple text-based user interface (UI) was created. In the future, a graphical user interface (GUI) could be integrated with the model to create a more user-friendly system for generating PDDL code from natural language.

Implementation Details

All of the code for prompt generation and processing of the LLM output was written in Python. Embeddings were computed using the 'sentence transformers' module, and the cosine similarity was computed using the 'sklearn' module. Data was stored in XML files, and was loaded and unloaded using the 'pickle' module. The 'nltk' module was used for POS tagging.

For testing and demonstration of the model, GPT-J-6B was used as the LLM, as this LLM can be called from a web-based UI, free of charge. In the future, the model can be integrated with GPT-3, and this might improve both the accuracy and the speed of the model.

4 Dataset

One of the main contributions of this research to the planning community was the creation of a novel dataset that consists of natural language annotations of durative actions in PDDL. By "annotation", what is meant is simply a natural language description of PDDL code. The PDDL code that was annotated is a set of durative actions from temporal PDDL domains. The temporal PDDL domains were obtained from Artificial Intelligence and Machine Learning Group, but only a subset of the durative actions found in these domains were annotated. The actions were annotated by members of the MuLab at Queen's University. Each annotator annotated every durative action in the dataset, and this resulted in several annotations for each action that differ in word choice and sentence structure.

The dataset that was created for the purposes of the present research could be useful to other researchers in the planning community. Therefore, it has been made freely available. To accompany the dataset, a datasheet containing descriptive information about the dataset has been created. Both the dataset and datasheet are available on Github, @QuMuLab/temporal-nl-to-pddl.

5 Evaluation

Determining whether the model is effective at converting NL input to temporal PDDL code is straightforward because the desired PDDL code can be known in advance. In fact, it was possible to test the model using NL from the dataset as test data. Obviously, using the same NL data for testing as is stored in the database would produce artificially good results, so this was not done. Instead, all annotations for a given durative actions were omitted from the database when the model was tested on this durative action. By following this method, annotations from the dataset could be used as input, and the prompts for the input could be generated from a database containing annotations for every durative action except the one corresponding to the current input. The model was tested on 4 of the 8 durative actions in the dataset, using annotations that were judged to be highly clear and detailed. The code that was generated by the model for each input is shown in Figure 2. Figure 1 shows an example of a prompt that was generated for the 'LIGHT_MATCH' action, and also shows an example of output for this prompt.

Prompt:
NL: At the start the param01 pred02 the source. At the end the param01 pred02 the param03.
PDDL: (and (at start (car-clear ?param01)) (at start (pred01 ?param03)) (at start (pred02 ?param01 ?param02)))
NL: The param13 must be at the param11, and the initial and param12s must be pred12. The param14 must be put pred13 and the param12 must be pred14. At the start of the action the param13 is no longer at the param11, and the param11 is pred14.
PDDL: (and (at start (pred11 ?param13 ?param11)) (at start (pred12 ?param14 ?param11 ?param12)) (at start (pred13 ?param14)) (at end (pred14 ?param12)))
NL: The initial and param22 are pred22. PDDL: (and (at start (pred21 ?param23 ?param21)) (at start (pred22 ?param24 ?param21 ?param22)) (at start (pred23 ?param24)) (at end (pred24 ?param22)))
NL: At the start the param31 and param32 must pred31 the param33. The param32 must be pred32, pred33, and pred35. The param31 must be pred34. At the beginning of the action the param31 will not be pred34. At the end of the action, the param32 will be not pred35 and the param31 will be pred34 again.
PDDL: (and (at start (at ?param32 ?param33)) (at start (at ?param31 ?param33)) (at start (pred32 ?param32)) (at start (pred34 ?param31)) (at start (pred33 ?param32)) (at start (pred35 ?param32)))
NL: The param41 is pred42. The param42 is pred44. The param42 is pred43, and pred45. When the action starts, the param41 is no longer pred44 and the param42 is not pred45. When the action is complete, the param42 is not pred45 and the param41 is pred44 again.
PDDL: (and (at start (at ?param42 ?param43)) (at start (at ?param41 ?param43)) (at start (pred42 ?param42)) (at start (pred44 ?param41)) (at start (pred43 ?param42)) (at start (pred45 ?param42)))
NL: At the start, the param51 has the first param52, and the second param52 must be pred52 throughout the action.
PDDL: (and (at start (robot-has ?param51 ?param52)) (over all (pred52 ?param52)))
NL: At the start the param61 pred61 the param63 where the param62 is as well. The param61 must be pred64 and the param62 must be pred62. The param62s will be pred63 as they are pred65. Doing this action means that the param61 will not be pred64 any more but by the end the param62 will not be pred65. Once the param62 gets untapped the param61 will be pred64 once again.
PDDL: (and (at start (at ?param62 ?param63)) (at start (at ?param61 ?param63)) (at start (pred62 ?param62)) (at start (pred64 ?param61)) (at start (pred63 ?param62)) (at start (pred65 ?param62)))
NL: At the start of the action, the param00 is designated as pred02 and is pred01. At the start of the action, the param00 is designated as pred02 and is pred01. At the start of the action, the param00 is designated as pred02 and is pred01.
PDDL:

Result:
(and (at start (pred02?param00)) (at start (pred01?param00)))

Figure 1: A LLM prompt that was generated by the model, and the resulting output of GPT-J-6B.

As shown in Figure 2, the model performed very well on relatively simple actions with clear and detailed NL input. The majority of the condition and effect statements were correctly extracted in the first three actions tested, although a few minor errors did occur. Since 'Action 4' was more

Action 1

Original code:

```
(:durative-action LOAD-TRUCK
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :duration (= ?duration 2)
    :condition (and (over all (at ?truck ?loc))
                    (at start (at ?obj ?loc)))
    :effect (and (at start (not (at ?obj ?loc)))
                            (at end (in ?obj ?truck)))
)
```

NL input: An action to load an object into a truck given the object, truck, and loading location. The truck and object both need to be at the loading location at the start of the action. After the action, the object is no longer at the location and is in the truck. The duration of this action is 2 minutes.

Generated code:

```
(:durative-action LOAD-TRUCK
    :parameters (?object ?truck)
    :duration (= ?duration 2)
    :condition (and (at start (at ?truck))
                    (at start (at ?object)))
    :effect (and (at start (not (at ?object)))
                 (at end (in ?object)))
)
```

Action 2

Original code:

```
(:durative-action change-color
    :parameters (?r - robot ?c - color ?c2 - color)
    :duration (= ?duration 5)
    :condition (and (at start (robot-has ?r ?c))
                               (over all (available-color ?c2)))
    :effect (and (at start (not (robot-has ?r ?c)))
                         (at end (robot-has ?r ?c2)))
)
```

NL input: An action that changes the colour currently being used by a painting robot given the robot, the current colour used by the robot c, and the new color to be used c2. The action requires that the robot is currently equipped with colour c and that the new color c2 is available for use. After the action, the robot no longer has color c and is now equipped with colour c2. The action takes 5 minutes to complete.

Generated code:

```
(:durative-action change-color
    :parameters (?robot ?c ?c2)
    :duration (= ?duration 5)
    :condition (and (at start (equipped ?robot ?c)
                    (at start (available ?c2))
    :effect (and (at start (not (equipped ?c)))
                 (at end (available ?c2)))
)
```

Action 3

Original code:

```
(:durative-action LIGHT_MATCH
    :parameters (?match - match)
    :duration (= ?duration 5)
    :condition (and (at start (unused ?match)))
    :effect (and (at start (not (unused ?match)))
                 (at start (light ?match))
                 (at end (not (light ?match))))
)
```

NL input: An action to light a match given the match to light. The action requires that the match to light is currently unused. At the start of the action, the match is designated as used and is lit. When the action is completed, the match is no longer lit. The action takes 5 minutes to complete.

Generated code:

```
(:durative-action LIGHT_MATCH
    :parameters (?match)
    :duration (= ?duration 5)
    :condition (and (at start (unused ?match ?match)))
    :effect (and (at start (used ?match))
                 (at end (not (lit ?match))))
)
```

Action 4

Original code:

```
(:durative-action untrap
    :parameters (?V - fire_brigade ?P - acc_victim
                 ?A - accident_location)
        :duration (= ?duration 25)
    :condition (and (at start (at ?P ?A))
                    (at start (at ?V ?A))
                    (at start (certified ?P))
                    (at start (available ?V))
                    (at start (waiting ?P))
                    (at start (trapped ?P)))
    :effect (and (at start (not (available ?V)))
                 (at end (not (trapped ?P)))
                 (at end (untrapped ?P))
                 (at end (available ?V)))
)
```

NL input: An action that takes a fire brigade to perform a rescue, an accident victim who is currently trapped, and an accident location to rescue them from. The action requires that both the fire brigade and the victim are at the accident location and that the victim is certified for rescues. The action also requires that the fire brigade is not currently rescuing somebody else and that the victim is currently waiting for rescue. At the start of the action, the fire brigade is designated as unavailable. At the end of the action, the victim is no longer trapped and the fire brigade is once again available for rescue operations. The action takes 25 minutes.

Generated code:

```
(:durative-action untrap
    :parameters (?victim ?location ?brigade)
    :duration (= ?duration 25)
    :condition (and (over all (trapped ?victim ?location))
                    (over all (at ?brigade ?location))
                    (over all (waiting ?victim ?brigade)))
    :effect (and (at start (unavailable ?brigade))
                 (at end (not (rescuing ?victim))))))
)
```

Figure 2: The code generated by the model on the 4 actions used for testing. The original (correct) code, and the NL input are also shown for each action.

complex than the first three actions, it was expected that the model would have more difficulty with this action. Indeed, the model was unable to generate all of the correct condition and effect statements for this action, although it made reasonable guesses. This shows that there is still work to be done to improve the performance of the model, especially on input that describes actions with many predicates. However, even though the model could be improved in many ways, the results are still satisfactory.

6 Related Work

The prospect of automatically inducing PDDL domains directly from natural language has interested many researchers. In fact, this task is widely regarded as an important step in making the application of automated planners more feasible in real world situations, and has been described as a bottleneck problem [2]. This task has the potential to save subject matter experts and knowledge engineers significant time and effort, and could increase the feasibility of using automated planning to solve problems in the real world [3].

One attempt to induce PDDL domains from natural language representations is found in [3]. This approach involves transforming natural language sentences into reduced representations (action templates), clustering these representations based on similarity, extracting consistent formulations

of the original sentences, and then inducing an appropriate PDDL domain using LOCM2. This process was not extended to handle natural language describing durative actions, but it was effective at generating formalisms from several natural language representations [3].

A similar approach was taken by Miglani, who attempted one-shot learning of PDDL domains from natural language process manuals using a pipeline with multiple stages. These stages included action sequence extraction using EASDRL, and domain induction using LOCM2. Miglani was successfully able to induce syntactically valid domains that captured most of the important information from the process manuals. Miglani also successfully predicted the durations of durative actions by extracting time phrases using spaCy's Named Entity Recognition (NER) [5].

Sil and Yates also attempted the similar task of extracting STRIPS representations of actions described in natural language text [7]. To accomplish this task, they used supervised learning along with common natural language processing (NLP) tools such as semantic role labelers, coreference resolvers and large-corpus statistics. In their discussion of possible future work, they mention the possibility of extracting representations for more complex actions, like durative actions.

In summary, significant progress has been made in extracting formal representations of planning domains from natural language. The prospect of extracting more complex representations from more diverse natural language input is an important continuation of work in this area.

7 Summary

The present research has revealed the great potential of using LLMs for code generation. The model developed in this research uses a very small database to generate LLM prompts, but still shows promising results. The results are in line with initial expectations, but the ease with which PDDL code can be obtained from a LLM was surprising. With further work, the performance of the model on descriptions of actions with many predicates could be improved, and errors could be minimized in general.

There are many ways that the model could be improved in the future. Firstly, if a larger database were used, there would be a higher likelihood of finding sentences in the database that are highly similar to input sentences. Moreover, there are many implementation details that could be modified in an attempt to improve the model's performance. For example, the amount of NL text and PDDL code used in a single LLM prompt could be modified to optimize performance. Another possible line of future work could involve the integration of more advanced techniques for parameter and predicate extraction. The human-in-the-loop system that is currently used for parameter and predicate extraction is sufficient for many purposes, but automatic extraction of parameters and predicates would greatly improve the capacity of the model to operate on unseen data without the need for any oversight.

The present research contributes to the field of automated planning in several ways. Firstly, the novel dataset that was created for the purposes of this research can be used by other researchers in the planning community. Secondly, the model developed proves that prompting of LLMs can be an effective method for generating PDDL code, and reveals the kind of prompt that can accomplish this. Finally, the research effectively generates some condition and effect statements from NL descriptions of these statements, and this constitutes an important step towards the goal of generating complete temporal PDDL code from natural language.

References

[1] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains, 2003.

[2] S. Kambhampati. Model-lite planning for the webage masses: The challenges of planning with incomplete and evolving domain models, 2007.

[3] A. Lindsay, J. Read, J. Ferreira, T. Hayton, J. Porteous, and P. Gregory. Framer: Planning models from natural language action descriptions, 2017.

[4] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl—the planning domain definition language, 1998.

[5] S. Miglani and N. Yorke-Smith. Nltopddl: One-shot learning of pddl models from natural language process manuals, 2020.

[6] D. Nau. Current trends in automated planning, 2007.

[7] A. Sil and A. Yates. Extracting strips representations of actions and events, 2011.