NL2LTL – A Python Package for Converting Natural Language (NL) Instructions to Linear Temporal Logic (LTL) Formulas

Francesco Fuggitti[1,2] · **Tathagata Chakraborti**[3]

[1] Sapienza University, Rome (Italy), [2] York University, Toronto (Canada)

[3] IBM Research, Cambridge (USA)

Primary Contact: fuggitti@diag.uniroma1.it

Abstract

This is a demonstration of our newly released Python package NL2LTL which leverages the latest in natural language understanding (NLU) and large language models (LLMs) to translate natural language instructions to linear temporal logic (LTL) formulas. This allows direct translation to formal languages that a reasoning system can use, while at the same time, allowing the end-user to provide inputs in natural language without having to understand any details of an underlying formal language. The package comes with support for a set of default LTL patterns, corresponding to popular DECLARE templates, but is also fully extensible to new formulas and user inputs. The package is open-source and is free to use for the AI community under the MIT license.

Open Source: https://github.com/IBM/nl2ltl

Video Link: https://bit.ly/3dHW5b1

1 Natural Language and LTL

A host of enterprise applications revolve around the management of workflows – this includes data processing pipelines in AutoML (**?**), web service composition (**?**), dialogue trees in conversational systems (**?**), and so on. An emerging theme in this area is the adoption of natural language as a desired input modality (**?**), aimed at reducing the barrier of entry and expertise required for users looking to adopt workflow management tools.

One of the scientific advances towards easier authoring tools for workflow management is the notion of *"declarative specifications"*. An important specification language in this paradigm is LTL (**?**), which allows for compilations to many well-known problems in process management e.g. conformance checking (**??**), while also admitting translations to specifications of standard reasoning engines such as automated planners (more on this later in Section 3).

While allowing for a declarative design paradigm, LTL also has a secondary benefit: LTL formulas can be readily described in an easy-to-understand natural language format. For example, DECLARE templates (**?**) – a very popular specification language in the world of business process management – is readily translatable to LTL formulas (**?**); and there exists a variety of tools to generate a human-readable construct given an LTL formula (**?**). In this work, we aim to make the journey in the opposite direction – from natural language to LTL. This has two main advantages: 1) unstructured inputs to a system can be translated to a form that reasoning engines can consume; while 2) the interface to the end-user remains as accessible as possible.

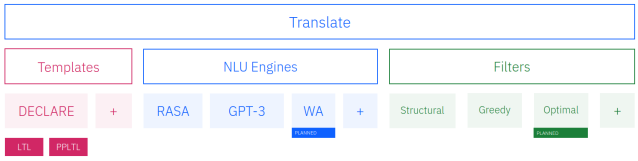

Figure 1: NL2LTL components

Related Work Existing works fall under two buckets: one which admits support for a range of LTL formulas but compromises on the expressiveness of the input (**????**), and the other which admits natural language inputs but were built for a particular domain like robotics and are not readily useful as a general purpose package for practitioners (**??????**). Furthermore, among these works, other than (**???**), none have publicly available code and are therefore not readily usable for practitioners. On the other hand, there are a couple of code bases that have also attempted natural language to LTL translation (**??**) but they are at a very rudimentary stage with little to no support or documentation. To the best of our knowledge, our package is the first one going public with support for a significant breadth of LTL patterns and an extensible API to make it usable in different domains.

2 NL2LTL Overview

NL2LTL is built with a unique focus on extensibility: 1) The inputs and outputs are domain agnostic so it can be adopted into any domain of choice; and 2) any and all components – be it the natural language understanding module or the scope of supported LTL formulas – are extendable or modifiable.

Sample Interactions Figure 3) illustrates the NL2LTL output – a list of translations, ranked by confidence. Each candidate translation has two interesting properties: **1) Explanations:** Each formula is translated back to English to illustrate how the formal representation interprets it. Depending on the downstream application, the developer can use this to explain to the end user to what extent the translation

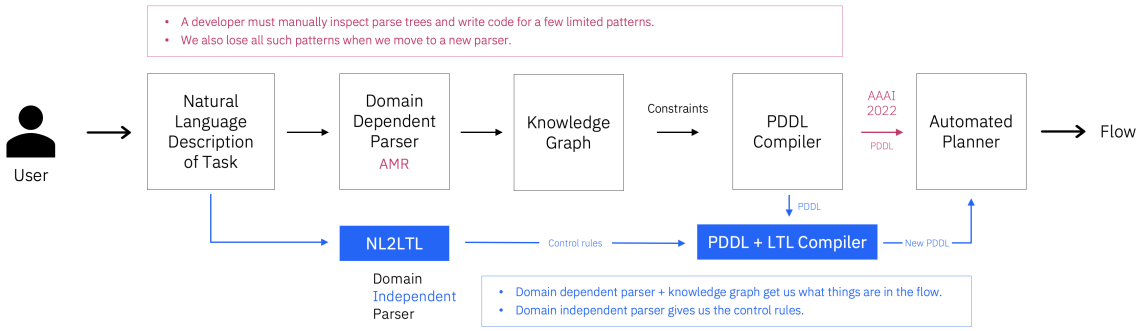

Figure 2: An application of NL2LTL to a real industrial application (**?**).

```
DECLARE Template: (ChainResponse Slack Gmail)
English meaning:  Every time activity Slack happens, it must be
                  directly followed by activity Gmail.
Confidence:       0.9999997615814209

DECLARE Template: (ExistenceTwo Slack)
English meaning:  Slack will happen at least twice.
Confidence:       1.0441302578101386e-07

DECLARE Template: (RespondedExistence Slack Gmail)
English meaning:  If Slack happens at least once then Gmail has
                  to happen or happened before Slack.
Confidence:       6.342362723898987e-08
```

Figure 3: Sample output (pretty print) of NL2LTL, illustrating candidate DECLARE templates suggested by the package, using the Rasa NLU Engine, for the request: *"Send me a Slack after receiving a Gmail"*.

matches their original request; and **2) Alternative Representations:** While this example if for DECLARE templates, each candidate can also be translated to other equivalent representations, such as LTL_f (**?**) and PPLTL (**?**).

Architecture The primary function of NL2LTL is a translation function, which is assisted by three different components (Figure 1). The first component is a set of supported patterns, or "templates", to be identified e.g. DECLARE templates (**?**). While this covers a wider range of LTL formulas, a developer can also add their own templates specific to their application. Secondly, NL2LTL can be configured with different NLU engines while the API remains exactly the same. At the moment, NL2LTL comes with two engines pre-configured – one based on the intent-entity paradigm from Rasa (**?**) that is traditionally used for natural language understanding, while the other is a language model-based extraction, tapping into the Open AI API (**?**). Finally, the package also implements a filter functions to post-process for better candidate sets of translations. This is because the patterns are not independent: 1) two LTL formulas can conflict with each other when both cannot be true at the same time; or 2) one LTL formula can be subsumed by another when the latter always implies the former.

3 Demonstration Logistics

The proposed demonstration at AAAI will consist of two live components – first the package itself, and then an illustration of how it can be employed in a real-world application.

Demonstration of NL2LTL For demonstration of the package itself, the AAAI audience will be able to interact with a command line interface where they can type in constraints in English, and explore the resultant LTL-translations along with the entities detected and what those translations mean when translated back to English.

Industry Application The second part of the demonstration will illustrate how the package is adopted for use in a real-world industry-scale application. For this, we will build on a system demonstration at AAAI 2022 (**?**) on a web service composition task using natural language. The previous system was dependent on code that is specific to the parser in order to look for specific patterns requested by the user – this means that there is a significant developer overhead in manually investigating the parser (Abstract Syntax Representations (**?**)) for a set of inputs, writing pattern extraction methods for it limited to that scope, while eventually that code is not reusable once the system upgrades to a different parser.

By augmenting the processing pipeline with the NL2LTL package, we are able to remove all parser-specific code and instead generate the required patterns with zero maintenance overhead. The patterns detected using the package, parallel to the original processing pipeline, are eventually merged for a singular PDDL input to the automated planner in the end, using the compilation which receives as input the PDDL specification generated by the original pipeline along with the LTL patterns detected by the NL2LTL package, and produces a compiled PDDL for the planner where the control rules are enforced (Figure 2). For the LTL to PDDL compilation, we use (**?**) but any existing approach (**????**) will suffice.

4 Acknowledgments

Francesco started and completed most of this work as an intern at IBM Research. Francesco was also partially supported by the ERC Advanced Grant WhiteMech (No. 834228), the EU ICT-48 2020 project TAILOR (No. 952215), the PRIN project RIPER (No. 20203FFYLK), and the JPMorgan AI Faculty Research Award "Resilience-based Generalized Planning and Strategic Reasoning".