Towards Automatic Composition of ASP Programs from Natural Language Specifications

Manuel Borroto Santana , **Irfan Kareem** and **Francesco Ricca**
Department of Mathematics and Computer Science, University of Calabria, Rende CS, 87036, Italy
{manuel.borroto,irfan.kareem,francesco.ricca}@unical.it

Abstract

This paper moves the first step towards automating the composition of Answer Set Programming (ASP) specifications. In particular, the following contributions are provided: (i) A dataset focused on graph-related problem specifications, designed to develop and assess tools for ASP automatic coding; (ii) A two-step architecture, implemented in the NL2ASP tool, for generating ASP programs from natural language specifications. NL2ASP uses neural machine translation to transform natural language into Controlled Natural Language (CNL) statements. Subsequently, CNL statements are converted into ASP code using the CNL2ASP tool. An experiment confirms the viability of the approach.

1 Introduction

Natural Language Processing (NLP) methodologies [Jurafsky and Martin, 2009] have determined impressive changes in the way we engage with computers, streamlining a multitude of tasks that would otherwise demand considerable time and proficiency from users. A noteworthy example is the emergence of tools for the automatic composition of computer programs [Ernst and Bavota, 2022; Peng *et al.*, 2023], like GitHub Copilot [Dakhel *et al.*, 2023; Kalliamvakou, 2022]. Nevertheless, prevailing tools of this nature predominantly cater to widely used programming languages, and limited (or no support) is available for many problem-solving formalisms in the area of Knowledge Representation and Reasoning (KR&R). An example of a formalism for KR&R that misses automatic program composition tools is Answer Set Programming (ASP). Answer Set Programming (ASP) [Brewka *et al.*, 2011; Gelfond and Lifschitz, 1988] is a declarative programming paradigm that can be used to solve complex AI problems. ASP originates from research on logic programming, nonmonotonic reasoning and knowledge representation. ASP became popular for featuring both an expressive declarative language and some efficient implementations [Gebser *et al.*, 2020], such as Clingo [Gebser *et al.*, 2016] and DLV [Alviano *et al.*, 2017]. ASP has been applied both in academia and industry, and it was effective in several knowledge-intensive applications of AI, such as scheduling, product configuration, robotics, workforce management, and decision support [Erdem *et al.*, 2016; Gebser *et al.*, 2020]. In the last few years, quite some effort has been spent on providing programming environments and tools for assisting programmers in devising ASP specifications [Vos *et al.*, 2012; Alviano *et al.*, 2023; Hahn *et al.*, 2023; Febbraro *et al.*, 2011; Busoniu *et al.*, 2013], including advanced editors, debuggers, testing frameworks, visualization tools, etc. Nonetheless, coding in ASP still is very challenging for beginners, especially for those having a weak mathematical and logical background, and is a time-consuming (sometimes repetitive) task also for expert programmers. On the other hand, once we look for the latest tools that significantly boosted the productivity of developers using mainstream imperative programming languages, it is easy to observe that a great benefit has been brought by the introduction of automatic program composition tools, such as GitHub Copilot [Dakhel *et al.*, 2023; Kalliamvakou, 2022]. However, no programming environment for ASP (as far as we know) features automatic code composition from the user's requests in natural language. Streamlining KR formalism with an automatic tool shows promise in boosting productivity for experts and easing entry for newcomers, potentially reducing adoption barriers, even in industrial settings. One might observe that generative AI tools, like ChatGPT, are capable of generating some ASP code snippets from specific prompts. Actually, they have been proficiently used to produce factual statements in an ASP-based system for reasoning from text [Yang *et al.*, 2023]; but, it can be easily verified that these AI models are not robust in generating valid and correct ASP programs. Thus, the problem of (robustly) composing ASP programs from statements provided in natural language remains open.

In this paper, the first steps towards filling this gap are taken, and the following contributions are provided:

(i) A dataset focused on graph-related problem specifications, that is conceived to develop and assess NLP tools for the automatic composition of ASP programs;

(ii) A two-step architecture, implemented in the NL2ASP tool, aiming at automating the translation of natural language specifications into ASP programs;

(iii) An experiment providing empirical evidence that the dataset and the architecture are a good first step towards automated code generation for ASP.

NL2ASP processes the input in two distinct phases: The

first phase draws inspiration from Neural Machine Translation (NMT). NMT [Yang *et al.*, 2020] methods utilize deep learning algorithms to translate text from one language to another, providing more natural and accurate results compared to traditional machine translation methods [Stahlberg, 2020]. NL2ASP employs NMT techniques to translate the input sentences in natural language to statements conforming to a recently-introduced controlled natural language for ASP [Caruso *et al.*, 2024]. For this task, NL2ASP can resort either to BART-base [Lewis *et al.*, 2019] or T5-small [Raffel *et al.*, 2020]. In the second phase, the Controlled Natural Language (CNL) statements are converted into ASP code using the CNL2ASP tool [Caruso *et al.*, 2024]. The performance of NL2ASP is measured in an experiment in terms of the quality of the statements produced in each of the steps of the composition process. In particular, the CNL statements produced by NL2ASP are evaluated in terms of BLEU [Papineni *et al.*, 2002], METEOR [Banerjee and Lavie, 2005] and Syntax Correctness Accuracy (a measure of the capability of producing syntactically-valid CNL statements). Eventually, the overall behaviour of the system is assessed by implementing an end-to-end test in which the capability of producing correct ASP specifications is measured. The obtained results are very promising and confirm the viability of the approach.

2 Related Work

The benefits and drawbacks of automated composition of programs have been studied in the literature [Ernst and Bavota, 2022], and the enormous potential of these tools is nowadays recognized [Kalliamvakou, 2022; Peng *et al.*, 2023; Dakhel *et al.*, 2023]. Almost all mainstream programming languages have been supported by academic or industrial tools [Chen *et al.*, 2021]. The benefits of providing tools for easing the development of ASP programs reducing the impedance mismatch existing from natural language specifications and ASP source code has been also recognized in the literature [Erdem and Yeniterzi, 2009; Fang and Tompits, 2017; Schwitter, 2018; Caruso *et al.*, 2024]. We are not currently aware of ASP-specific tools targeting automatic program composition from natural language based on NMT as NL2ASP. Nonetheless, there are some that support inputs from more structured or simpler languages.

Baral et al. automated the solving of logic puzzles in simplified English by translating their descriptions into ASP [Baral and Dzifcak, 2011] by resorting to λ-calculus and probabilistic combinatorial categorical grammars. The Controlled Natural Languages (CNLs) are subsets of the full natural languages with restricted grammar and vocabulary [Kuhn, 2014]. Several efforts have been made to develop CNLs that target ASP programs. Erdem et al. described the specific grammatical structure of a CNL named BIOQUERYCNL, as well as the algorithm for converting queries into ASP [Erdem and Yeniterzi, 2009]. Fang et al. introduced a CNL approach for representing answer sets based on Language for ANnotating Answer-set programs (LANA) annotations [Fang and Tompits, 2017] which was implemented in the SeaLion IDE. Schwitter developed in 2018 a grammar to specify and verbalize answer set programs using a CNL named PENG^{ASP} [Schwitter, 2018]. Lifschitz [Lifschitz, 2022] highlighted the connection between mathematical definitions and the knowledge representation capability of ASP. More recently, Dodaro et al. presented CNL2ASP, a comprehensive publicly-available tool that converts controlled natural language into ASP programs [Caruso *et al.*, 2024]. NL2ASP resorts to the CNL and tool devised by Dodaro et al. CNLs ease the task of programming but still require human developers. NL2ASP aims at overcoming this by making the encoding process automatic. Our approach is also related to the development of datasets for knowledge-based question answering and to other applications of ASP to natural language. Datasets that are useful for question answering were proposed by Perevalov et al. that extended the Knowledge Graph Question Answering (KGQA) benchmarks QALD-9 by adding question query pairs up to 8 languages [Perevalov *et al.*, 2022]. Dubey et al. introduced a comprehensive dataset for complex question answering, called LC-QuAD 2.0 [Dubey *et al.*, 2019]. This dataset includes 30,000 questions, their corresponding paraphrases, and SPARQL queries. An NMT method based on long short-term memory (LSTM) units was presented by Sutskever et al. on an English-to-French translation task [Sutskever *et al.*, 2014]. As part of the English-to-French translation task, Bahdanau et al. proposed an NMT module based on Bidirectional Recurrent Neural Networks [Bahdanau *et al.*, 2015]. An NMT method for KBQA that automatically translates natural language in SPARQL queries was proposed by Borroto and Ricca [Borroto and Ricca, 2023] that performed well in the Question Answering over Linked Data (QALD) competition. More recently, Nye et al. presented a GPT-3 based dual-system (neural System 1 and the logical System 2) model, which generates the semantic parses from natural language sentence and couples it with reasoning modules [Nye *et al.*, 2021]. In this line of research, Yang et al. proposed LLMs like GPT-3 can act as few-shot semantic parsers, converting natural language into logical forms for answer set programming (ASP) [Yang *et al.*, 2023]. This system can address diverse question-answering tasks without distinct retraining. Ishay et al. leveraged LLMs to get the ASP with prompt engineering for solving logic puzzles [Ishay *et al.*, 2023]. They have used the logic puzzle dataset of [Mitra and Baral, 2016]. Although LLMs excel in System-1 processes, their results can frequently be inconsistent and incoherent. Mitra and Baral proposed a heterogeneous agent model for question-answering tasks in Facebook's bAbl dataset [Mitra and Baral, 2016] where ASP is used for knowledge representation and reasoning. Moldovan et al. demonstrated that a logic prover can be integrated into a Question Answering system [Moldovan *et al.*, 2003], where logic representations are used to transform questions and answers.

3 Basics on Answer Set Programming

Answer Set Programming (ASP) [Brewka *et al.*, 2011; Gelfond and Lifschitz, 1991] is a declarative logic-based language for knowledge representation and reasoning. Rather than writing algorithms, the ASP programmer describes problems with a declarative formal language and uses an ASP sys-

tem [Lierler *et al.*, 2016] to find solutions [Lifschitz, 2019].

The Language of ASP. The main construct of the ASP language is the logic rule, which is a construct of the form:

$$a_1 \mid \ldots \mid a_n \; :- \; b_1, \ldots, b_k \; ; \; not \; b_{k+1}, \ldots, \; not \; b_m \quad (1)$$

where a_i ($0 \leq i \leq n$) are *atoms*, b_i ($0 \leq i \leq k \leq m$) and $not\; b_j$ ($k \leq j \leq m$) are positive and negative *literals*, respectively. Informally, a rule reads as follows: "at least one of the a_i is true whenever all b_i ($0 \leq i \leq k \leq m$) are true and all b_j ($k \leq j \leq m$) are false". On the right-hand side of a rule is the *body* (or antecedent) and on the the left-hand side of a rule is the *head* (or consequent). The rule is called *Fact* if the body of a rule is empty. A rule with an empty head is called *Constraint*. An ASP program is a set of logic rules, which is interpreted according to common sense principles. An ASP program represents a problem to be solved that together with input (also expressed by a collection of rules) admits a collection of answers (possibly also no answer) corresponding to the solutions of the problem [Lifschitz, 2019].

For example, the well-known 3-colorability is solved by:

$$col(X, red) \mid col(X, green) \mid col(X, yellow) \; :- \; node(X)$$

$$:- col(X, C), col(Y, C), edge(X, Y)$$

The first rule reads "if X is a node then it can be colored either in red or in green or in yellow", and the second rule (a constraint) reads "it is not possible that adjacent nodes have the same color". Here, the input is modelled by a set of facts of the form $node(\cdot)$ and $edge(\cdot, \cdot)$. Suppose we provide as input the graph $node(1)$, $node(2)$, $node(3)$, $edge(1, 2)$ and $edge(1, 3)$; it can be verified that a possible solution (answer set) is $\{col(1, red), col(2, green), col(3, green)\}$. The 'color' is abbreviated as 'col'.

ASP also supports weak constraints, choice rules, aggregates, etc.; however, a full description of the ASP language is not instrumental for this paper. Please refer to dedicated literature [Calimeri *et al.*, 2020; Brewka *et al.*, 2011; Gelfond and Lifschitz, 1991; Lifschitz, 2019; Gebser *et al.*, 2012] for a more detailed (and formal) account on ASP.

Controlled Natural Language for ASP. Controlled Natural Languages (CNLs) are subsets of natural languages with restricted grammar and vocabulary [Kuhn, 2014].

A comprehensive CNL tool for ASP that supports all the main language constructs is CNL2ASP [Caruso *et al.*, 2024]. In CNL2ASP, CNL specifications are made up of propositions that are structured by means of clauses, connected by connectives to express concepts and conditions. To provide flavour and a basic understanding of CNL2ASP we resort to our running example, i.e., an encoding of the 3-colorability problem. First of all, we define node, edge and color by using the following definition statements:

```
A node is identified by an id.
A edge is identified by a firstnode, and by
a secondnode.
A color is identified by an id.
```

These are used by CNL2ASP to initialize the internal data structures and know about the domain of discourse. Then, we specify the assignment of colors to nodes by using the following `whenever/then` clause:

```
Whenever there is a node with id X then we
can have a col with node X, and with color
equal to blue, or a col with node X, and with
color equal to red, or a col with node X, and
with color equal to green.
```

Finally, we specify valid colorings with the following *negative strong constraint* statement:

```
It is prohibited that C1 is equal to C2,
whenever there is a col with node X, and with
color C1, whenever there is a col with node
Y, and with color C2, whenever there is an
edge with firstnode X, and with secondnode Y.
```

The ASP program resulting from a call to CNL2ASP is:

$$col(X, blue) \mid col(X, red) \mid col(X, green) \; :- \; node(X).$$

$$:- C1 = C2,\; col(X, C1),\; col(Y, C2),\; edge(X, Y).$$

It is worth observing that the CNL specification is amenable to a human and does not require to know the technicalities of the syntax of ASP. The interested reader can find more details CNL2ASP in the original article [Caruso *et al.*, 2024].

4 The Problem and Some Assumptions

The problem we tackle in this paper is to ease the process of composing formal ASP statements (ASP programs) from problem specifications given in natural language. However, this is quite an ambitious final objective. In order to take the first steps towards pursuing this goal we begin by making some pragmatic simplifying assumptions.

The first assumption we make is: problem formulations are given as a bag of statements, where contiguous subsets of statements can be encoded with a logical rule. This makes it possible to compose a program by iteratively transforming groups of statements in the corresponding rule.

The second assumption we make is that we are given a corpus of pairs that associates each bag of statements to a *"gold program"*. The gold program is an example of an expected encoding of the statements in ASP. A subset of that corpus can be used to *train* a neural network to perform the task of ASP program composition (training set); another (disjoint from the training set) subset of that corpus can be used for testing the correctness of the system (test set).

The problem we tackle is, thus, formalized as follows: Given a specification P of a problem (a bag of statements), compose a program P_{asp} that is uniform equivalent to the gold program P_{gold} associated to P.

Two ASP programs P_1 and P_2 are uniform equivalent iff for any set of (non-disjunctive) facts F, $P_1 \cup F$ and $P_2 \cup F$ admit the same answer sets [Eiter *et al.*, 2007]. This captures that an ASP program models uniformly a problem over varying instances provided via facts [Brewka *et al.*, 2011].

The formulation of the composition problem makes it possible to iteratively process the input by identifying the statements corresponding to a rule, and transforming such statements into the corresponding rule. In our approach, this latter task is performed in two steps: (i) translation of the natural language in a corresponding CNL (NMT task); (ii) conversion of the CNL in ASP (the basic task of CNL2ASP).

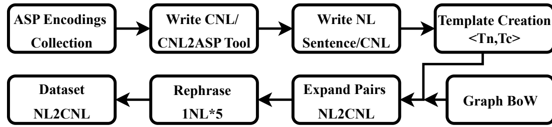

Figure 1: Dataset creation workflow

Sentence	Template
CNL: *Node 1* have an *edge node X*, where *X* is one of *2, 5*.	TCNL: *Noun_1 num_1* have an *verb_1 noun_1 var_1*, where *var_1* is one of *num_2, num_3*.
NL: There is *node 1* has an *edge* to *node X*, where *X* is one of the numbers *2 or 5*.	TNL: There is *noun_1 num_1* has an *verb_1* to *noun_1 var_1*, where *var_1* is one of the numbers *num_2 or num_3*.

Table 1: An overview of CNL & NL pair template

To the best of our knowledge, no dataset supporting the above tasks is available; thus, we have developed one as described in the next section. Pragmatically, we concentrated our efforts on graph-related problems. This is a domain of problems that is classically addressed with ASP [Brewka *et al.*, 2011; Erdem *et al.*, 2016; Lierler *et al.*, 2016; Calimeri *et al.*, 2020], and is very rich and variegate (in terms of number, complexity and importance of problems) to be considered a meaningful pragmatic choice to assess our approach.

Concerning the issue of testing our implementation, we observe that checking solutions is intractable (i.e., is undecidable in general and very hard for specific cases, e.g., uniform equivalence belongs to the second level of the polynomial hierarchy for already ground disjunctive ASP programs) [Eiter *et al.*, 2007]. Thus, in our experiment we will pragmatically consider, in analogy to what has been done in other similar contexts, looser measures of correctness, such as purely syntactic measures (BLEU, METEOR); and, in end-to-end tests, correctness was certified by a human expert.

5 Dataset Construction

One of the contributions of our work is a specialized dataset centered on graph-related problem specifications. This section details the creation process of this dataset, named NL2CNL. The workflow of dataset creation is illustrated in Figure 1, which entails collecting ASP encodings of graph-related problems. NL2CNL includes problem statements in natural language and corresponding CNL propositions.

The ASP encodings were mainly collected from ASP competitions (2015, 2017), books, lecture notes, data shared by renowned professors, and other online sources. For each collected ASP encoding, we have written the corresponding CNL and generated a new ASP encoding by using the CNL2ASP [Caruso *et al.*, 2024] tool. Then, we cross-verified the answer sets of both actual ASP encoding and tool-generated ASP encoding to identify trivial problems, and checked the encoding manually. This way, we ensure the CNL we have written is compliant with the actual encoding. We have tried to cover the five main grammar propositions of tool CNL2ASP: (i) negative strong constraint proposition, (ii) positive strong constraint proposition, (iii) weak constraint proposition, (iv) definition proposition, and (v) quantified choice proposition. We have ensured that each CNL is accompanied by a corresponding natural language statement that accurately reflects the original semantics, while also retaining important information about the *verb, noun*, and *variable names* used in the CNL. Despite successfully generating NL statements and CNL propositions for 20 programs, the resulting dataset (494 pairs) is too small for effective use in data-driven approaches. To address this limitation, augmentation techniques are applied to expand the size and diversity of the dataset.

Motivated by the existing template-based approaches [Hua and Wang, 2020; Dubey *et al.*, 2019; Xu *et al.*, 2018; Trivedi *et al.*, 2017] in creating quality datasets, we decided to apply this type of approach in our case. Templates ensure content consistency by generating syntactically and semantically correct data, minimizing dataset errors. They are easily modifiable and scalable, ensuring well-structured, error-free datasets with specific patterns. Thus, we manually created CNL-NL template pairs based on the propositions of CNL2ASP, and ended up developing 369 unique templates for each grammar proposition.

Templates were obtained as follows: we start from a valid CNL-NL pair, and we replace specific parts of speech (nouns, verbs, etc.) with placeholders in order to obtain a template CNL-NL pair. A placeholder is a symbolic representation of a specific replacement, and we used conventions for different categories: numbers are "num_X", verbs are "verb_X", nouns are "noun_X", variables are "var_X", colors are "color_X", and special predicate identifiers *PIDs* are "PID_X", and so on ("X" is a positive integer ensuring uniqueness of placeholders). Based on CNL patterns for specified facts, numeric placeholders are: (i) *num_range*, defined to signify a range of numbers; and (ii) *num_choice*, used for referencing specific numbers. For instance, the range from 1 to 4 is presented as *num_range (1 to 4)*, and *num_choice (1, 2, and 5)* or *num_choice (2 or 3)*. This process is exemplified in Table 1, where a template is obtained by introducing placeholders in sentences of a CNL-NL pair, i.e. *"noun"* placeholder, used to replace the word *"node"*, *"verb"* placeholder replaces the word *"edge"*, variable *"var"* placeholder replaces capital alphabet *"X"*, and number *"num"* placeholders replace the numbers *2 and 5*. Numeric placeholders are accompanied by other categories like verb *"verb_X"* and noun *"noun_X"* for word variations. Placeholder *"var_X"* represents capital alphabets, while *"col_X"* denotes a list of colors. Special placeholders are introduced for contexts with the clause "whenever,", where the defined list of placeholders follows the pattern of *PID_X*, e.g. *PID_1* is replaced with *"first vtx"*.

After creating the unique template CNL-NL pairs, we created suitable *Bag-of-Words* (BoW) to obtain possible replacements for placeholders. We considered three categories of words, *verbs, nouns*, and special predicate identifiers like *PIDs*. We manually identified: 21 *PIDs*, 77 *nouns*, and 408 *verbs*. Words that might be used as predicate names were taken from problem specifications, and the others were added so to construct meaningful CNL and NL.

Having defined the templates and the BoWs, we replaced the placeholders at random by respecting the placeholder expected type (e.g., *verb_X* in *templates*, it is replaced by associating a verb from the BoW of *verbs*), *num_X* are replaced by integer values, *num_range* is replaced by numbers gener-

Grammar Type	Source	Gener.	Rep.Ct.	Total
Def. Const/Comp.	154	21	875	1050
Def. 'When'	145	15	800	960
Def. 'Whenever'	110	41	755	906
Neg. Constraint	22	138	800	960
Pos. Constraint	39	121	800	960
Quant. Choice	13	156	845	1014
Weak Constraint	11	149	800	960
Grand Total	494	641	5675	6810

Table 2: Count of CNL2ASP propositions in NL2CNL. 'Gener.' stands for 'Generated', 'Rep.Ct.' stands for 'Rephrased Count'.

ated at random within the delineated limits, *num_choice*, we choose random integers between 1 and 10 and arranged them in order and comma separated.

The distinct counts for each grammar proposition is reported in Table 2 in column "Source". To mitigate potential class imbalances in training data-driven models, we instantiated the templates in such a way that the total number of NL2CNL pairs per proposition type in the expanded dataset is almost equal. In summary, we augmented the source dataset with 641 template-generated pairs (cfr. "Gener." column in Table 2), resulting in a total of 1135 NL2CNL pairs.

To enhance the quality and diversity of our dataset, we decided to perform an additional paraphrasing task on all the NL statements. This process increases the dataset size and introduces varied linguistic representations while maintaining the original meaning so to enable effective training of a robust Language Model (LM). Thus, we used *OpenAI API*[1] for this task, with *"text-davinci-003"* engine with specific parameters, such as: prompt set to *"Rephrase the following sentence: sentence"*, temperature value of 0.6, and maximum token limit of 1000. Each sentence was rephrased five times in order to strike a balance between diversity and semantic dilution. Generation of more than five may lead to unintended overlaps or even slight semantic shifts. After rephrasing natural sentences the count increased to 5675 (cfr. "Rep.Ct." column in Table 2). All available NL-CNL pairs constitute the dataset NL2CNL of 6810 pairs (see Table 2).

6 The NL2ASP Tool

The architecture of the NL2ASP tool for the automatic composition of ASP programs from natural language statements is depicted in Figure 2. In the first step, we transform NL statements into CNL statements, and in the second step, we obtain the ASP code by running the CNL2ASP tool. As a consequence, we inherit in the implementation some limitations of CNL2ASP, such as the fact that the user has to specify the schema (or the facts) before providing the rules, and the rule specifications must be provided in such a way that some body-to-head dependency is respected [Caruso *et al.*, 2024]. Note that, this limitation is not present in the translation from sentences to CNL. Indeed, in the proposed architecture, the first step relies on the recent advances in Neural Machine Translation [Yang *et al.*, 2020]. Specifically, encoder-decoder-based Transformer architectures are incorporated to

[1]https://platform.openai.com/docs/api-reference

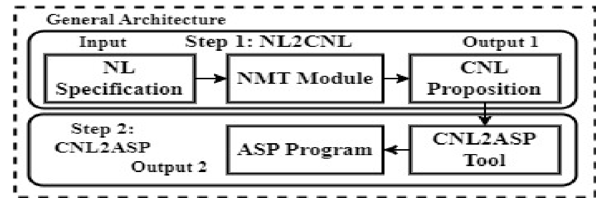

Figure 2: Architecture for automatic composition of ASP programs.

facilitate the translation of NL specifications into CNL propositions. Using these high-performance models, we capture language nuances and produce grammatically accurate translations. The use of NMT to address the first step is a decision that arises intuitively, as it is easy to realise that the task is like translating from English into another (more restricted) language, i.e., the language of CNL2ASP. Our architecture is simple, ensuring adaptability, can be implemented with different NMT tools, and can take profit from the dataset we have presented in the previous section.

In the implementation we used state-of-the-art transformer-based architectures i.e., T5 [Raffel *et al.*, 2020] and BART [Lewis *et al.*, 2019], which have proven their efficiency for different language tasks like text summarization, question-answering, and machine translation. The T5 model introduces an encoder-decoder transformer architecture that undergoes pretraining in a combination of unsupervised and supervised tasks, with each task transformed into a text-to-text format [Raffel *et al.*, 2020]. The BART model follows a denoising autoencoder approach, where the input sequence is corrupted using a noise function [Lewis *et al.*, 2019]. Then, the sequence of corrupted entries passes through a bidirectional encoder, which captures contextual information from both directions. Then, the encoder representation is introduced into an autoregressive decoder from left to right, which generates the sequence by predicting the next token based on the previously generated tokens.

7 Experiments

We have evaluated empirically our approach, and, in this section, describe two sub-analyses: (i) assessment of the NMT model's translation quality, and (ii) end-to-end assessment of the proposed architecture, i.e., assessment of the ability of our NL2ASP to accurately create ASP programs from natural language. To this end, we fine-tuned pre-trained versions of the T5 and BART models on the proposed dataset and measured several evaluation metrics. Datasets and tools are available at https://github.com/IrfanKareem/nl2asp.

Software and Hardware Details. We adopted the pre-trained T5 and BART models freely available on the Hugging Face platform (https://huggingface.co/). Specifically, we used the T5-small (https://huggingface.co/t5-small) and Bart-base (https://huggingface.co/facebook/bart-base) models, which have a size of 60 and 140 million of parameters, respectively. The choice of versions was based on the capabilities of the hardware at our disposal.

Split	BLEU-1	BLEU-2	BLEU-3	BLEU-4	MET
0	0.955	0.944	0.939	0.925	0.960
1	0.966	0.957	0.952	0.940	0.969
2	0.961	0.952	0.947	0.935	0.966
3	0.959	0.949	0.944	0.931	0.961
4	0.962	0.952	0.947	0.935	0.965
Avg.	0.961	0.951	0.946	0.933	0.964

Table 3: Scores across splits for T5-small model

Split	BLEU-1	BLEU-2	BLEU-3	BLEU-4	MET
0	0.841	0.783	0.751	0.686	0.875
1	0.845	0.786	0.753	0.686	0.877
2	0.853	0.797	0.767	0.704	0.890
3	0.818	0.758	0.726	0.660	0.864
4	0.842	0.787	0.757	0.696	0.879
Avg.	0.84	0.782	0.751	0.686	0.877

Table 4: Scores across splits for Bart-base model

The models were implemented using Keras 2.12.0 on top of Tensorflow and the Transformers 4.30.2 library. To run the experiments, we used an Ubuntu 20.04 server with 500GB of RAM, a 16 GB NVIDIA Tesla V100-PCIE GPU card, and CUDA 11.8. The code ran using Python 3.9.17 in a Jupyter Notebook environment.

Evaluation Measures. The translation quality of the model was evaluated using the BLEU [Papineni *et al.*, 2002], METEOR [Banerjee and Lavie, 2005], and the translation *Syntactic Accuracy (SA)* measures. BLEU is founded on precision-based attributes and functions by comparing the n-grams (a sequence of n words) found in the candidate translation with those in one or more reference translations. BLEU falls within the range of 0 to 1, the larger the higher degree of similarity between the translation and the reference translation. METEOR [Banerjee and Lavie, 2005] assesses machine translation hypotheses by aligning them with one or more reference translations. This alignment process considers exact, stem, synonym, and paraphrase matches between words and phrases. The metric relies on the harmonic mean of precision and recall for unigrams. A value closer to 1 indicates a higher translation quality. METEOR is said to have a better correlation with human judgment than BLEU. The SA (see Equation 2) metric, measures the proportion of translated sentences that conform to the CNL2ASP grammar.

$$SA = \frac{\#\text{syntantically correct sentences}}{\#\text{total sentences}} \tag{2}$$

7.1 Assessment of the NMT Models

Experiment_A: Cross-Validation. In the first experiment, we performed K-fold cross-validation, models T5 and BART are trained k times, the dataset is divided in one fold as a test set, and k-1 folds are used as the training set. This approach provides a robust estimate of model generalization ability. The value of k was set to 5. We use all 6810 NL to CNL pairs from NL2CNL dataset, which are named as features of *"sentences"* and *"targets"* respectively.

To perform the text tokenization, we used the *AutoTokenizer* utility provided by the Transformer library, which allows us to use the pre-trained tokenizers of T5-small and Bart-base. To handle the collating and batching of input data for training, we used the *DataCollator* utility in the Transformer library. It plays a vital role in streamlining the data preparation process and ensuring seamless integration during training and inference. The batch size was set to 16.

The Adam Weight Decay (AdamW) optimizer is used in the training process. This optimizer is based on the adaptive estimation of first and second order moments with an added method to decay the weights. AdamW is a variant of the Adam optimizer that decouples weight decay from the adaptive learning rate, which leads to better performance. We set learning rate= 2×10^{-5}, and weight decay= 0.01.

To calculate the BLEU we have used *"corpus_bleu"*, a function from the NLTK [Wagner, 2010] library in Python. It computes the BLEU score of the entire dataset, by taking multiple pairs of source and reference sentences. Moreover, we decided to calculate the cumulative BLEU up to 4-gram. We have tuned METEOR with standard parameters, i.e. alpha=0.9, beta=3, and gamma=0.5. These values make the trade-off between recall, precision, the penalty for sentence length mismatches, and word order differences respectively. We ran the training of the models for 200 epochs for each split and applied the Early Stopping technique to monitor the validation loss with minimum mode and patience equal to 20, thus reducing the overfitting and computational resources.

Table 3 illustrates the T5-small BLEU and METEOR scores by split and the final average. The BLEU scores for all n-grams are generally high, ranging from 0.90% to 0.97%, which shows strong performance in terms of n-gram overlap with the reference text. The METEOR score also indicates high performance, with values ranging from 0.94% to 0.98% across the splits. On the other hand, Table 4 shows the metrics scores for the fine-tuned Bart-base model. Bart-base model scores are low compared to model T5-small (cfr. Table 3).

Although training times might vary with hardware, there are significant differences in training and scoring times between T5-small and Bart-base models. T5-small takes about 5 hours on average to train across the split, with scoring times of 3 to 5 minutes, while Bart-base averages approximately two hours for training and six hours for scoring. Despite Bart-base requiring fewer epochs, it spends more time on calculating translation quality measures. T5-small surpasses Bart-base in all translation quality measures and excels in high-quality text generation. Despite a longer training time, T5-small delivers faster predictions compared to Bart-base. These performance advantages make T5-small the preferred choice, despite it requires longer training times.

Experiment_B: Syntax Correctness. This more demanding experiment aims at checking the syntactic correctness of the CNL propositions produced by the two models. Thus, we assessed each CNL proposition using the syntax checker from the CNL2ASP tool. In particular, the syntactic correctness checking was performed during K-fold cross-validation by running the trained models on their splits. Since the NL2CNL dataset has a length of 6810, for each split, a validation set of 1362 examples is used. Again model T5-small, with a SA of about 94%, performs better than model Bart-base with SA is

Split	Total	Bart-base		T5-small	
		Errors	Acc.	Errors	Acc.
0	1362	650	52.28	93	93.17
1	1362	630	53.74	78	94.27
2	1362	572	58.00	69	94.93
3	1362	619	54.55	82	93.98
4	1362	600	54.95	80	94.13

Table 5: Syntax check for T5-small & Bart-base with K=5.

Metric	T5-small	Bart-base
BLEU-4	0.860	0.704
METEOR	0.935	0.876
Precision	0.929	0.860
Recall	0.927	0.921
F1-Score	0.928	0.88
Accuracy	0.368	0.253

Table 6: Evaluation measures for T5-small & Bart-base.

around 55% (cfr. Table 5).

Looking at problematic generations we observe that T5-small exhibits issues, particularly with the clause “whenever” when used with other propositions. Also sentences exceeding 60 characters occasionally result in incomplete generation. Some minor issues are present in the transformation of “less than or equal to” to “at most or equal to,”. Nonetheless, a human user can fix these issues with minor adjustments. On the other hand, Bart-base model exhibits some more issues in addition to those previously discussed for T5-small. These include occasional generation of additional spaces (e.g., “C2” generated as “C 2”), inconsistent word connections (e.g., vice-versa), and problems with capitalization (e.g., “whenever There is a”). Furthermore, the model occasionally uses dashes instead of underscores to connect words (e.g., “Dominating_Set” vs. “Dominating-Set”), which is not acceptable for the CNL2ASP parsing tool, but could be addressed with post-generation processing.

Experiment_C: Assessment on Unseen. In this experiment, we train both models with the complete dataset. During training, T5 converged at 155 epochs, with a training loss of 0.022 and validation loss equal to 0.0467; the best BART model was obtained at 37 epochs, with training and validation losses of 0.010 and 0.0492, respectively. These values are quite low, evidencing the ability of the models to learn from the dataset. As test set we prepared a new test dataset manually, which consisted of 209 new specifications. We did the inference and calculated BLEU, METEOR, accuracy, precision, recall, and F1-score for both T5-small and Bart-base. T5-small consistently outperformed Bart-base in all metrics, see Table 6. Both models generalize well and achieve robust performance on unseen instances.

7.2 End-to-End Evaluation

We now measure the ability of the NL2ASP tool to produce valid ASP programs. To this end, we targeted six well-known graph-related problems, i.e. Maximize Clique, Hamiltonian Cycle, Graph Coloring, Connected Dominating Set, Traveling Salesman Problem, and Hierarchical Clustering. We manually created a dataset with the specifications for the aforementioned problems, including two different encodings for the Graph Coloring problem (called GCv1 and GCv2). The inherent flexibility of natural language allows for a multitude of ways to express the same concept. So to make the assessment more realistic the dataset was expanded by paraphrasing the former specifications with the support of ChatGPT. In total, we obtained 21 problem specifications.

The NL2ASP tool, configured with the best performing T5-small fine-tuned in the Experiment_C, was fed with one problem specification at time to obtain the corresponding ASP programs. Since automatic checking of correctness is unfeasible, we manually checked the produced programs.

During the translation step, NL2ASP was able to produce 389 syntactically correct CNL statements out of 393, performing for a good 98.98%. In this case, we found minor errors, for example, our tool generated *“at most or equal to”* instead of *“less than or equal to”*. The mentioned issues were found in two variants of the Hierarchical Clustering problem, so NL2ASP was able to produce 19 *correct* ASP programs over 21 problem specifications. It is important to mention that only minor manual fixes would be required by a human to fix the aforementioned issues that affected Hierarchical Clustering instances.

Additional Experience With LLMs. As a final test, we wanted to check to what extent a general purpose LLM, such as GPT 3.5, is capable of carrying out ASP program composition. The goal of this experiment is not to establish a direct comparison with ChatGPT, but to ensure that specific-purpose tools like ours can perform better than general-purpose models that are not directly trained to tackle this kind of tasks. The prompt we used to request the program generation is: *“Please give the ASP program of the following natural language statements:”*, followed by the natural language statements. As expected, the “out-of-the-box” ChatGPT was able to generate ASP programs given the specification. However, these programs are neither syntactically nor semantically correct most of the time (17/21), and quite some effort would be required to fix them manually. For all wrong programs we asked ChatGPT to regenerate them 3 times, but this resulted in no gain. It is an open research question to explore how/if these models can be robustly used for this task.

8 Conclusion

This paper introduces the NL2CNL dataset and NL2ASP tool for the automatic composition of ASP programs. NL2ASP is based on a two-step architecture, which transforms natural language specifications in CNL statements that, eventually, are converted into valid ASP programs. NL2ASP has been implemented with two Transformer-based models for NMT tasks, i.e. T5-small, and Bart-base. In our experiments, T5-small performed better than Bart-base for text-to-text translation according to several translation quality measures. Our future work involves expanding the dataset to include non-graph problems, enhancing both the dataset and system to process more natural sentences with reduced reliance on CNL2ASP constraints and explicit mention of variables, and further exploring the usage of LLM for this task.

Acknowledgments

This work was supported by Italian Ministry of Research (MUR) under PRIN project PINPOINT, CUP H23C22000280006, PNRR projects FAIR "Future AI Research" - Spoke 9 - WP9.1 - CUP H23C22000860006, Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and by Italian Ministry of Enterprises and Made in Italy (MIMIT) under project EI-TWIN "Concezione, progettazione e sviluppo di soluzioni di Energy digital TWIN" Prog n. F/310168/05/X56 - CUP: B29J24000680005 - COR: 22338845.

References

[Alviano *et al.*, 2017] Mario Alviano, Francesco Calimeri, Carmine Dodaro, Davide Fuscà, Nicola Leone, Simona Perri, Francesco Ricca, Pierfrancesco Veltri, and Jessica Zangari. The ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, pages 215–221. Springer, 2017.

[Alviano *et al.*, 2023] Mario Alviano, Davide Cirimele, and Luis Angel Rodriguez Reiners. Introducing ASP recipes and ASP chef. In *ICLP Workshops*, volume 3437 of *CEUR WP*. CEUR-WS.org, 2023.

[Bahdanau *et al.*, 2015] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

[Banerjee and Lavie, 2005] Satanjeev Banerjee and Alon Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*, pages 65–72. ACL, 2005.

[Baral and Dzifcak, 2011] Chitta Baral and Juraj Dzifcak. Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation. In *AAAI Fall Symposium: Advances in Cognitive Systems*, volume FS-11-01 of *AAAI Technical Report*. AAAI, 2011.

[Borroto and Ricca, 2023] Manuel A. Borroto and Francesco Ricca. Sparql-qa-v2 system for knowledge base question answering. *Expert Syst. Appl.*, 229(Part A):120383, 2023.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[Busoniu *et al.*, 2013] Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky, and Hans Tompits. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory Pract. Log. Program.*, 13(4-5):657–673, 2013.

[Calimeri *et al.*, 2020] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.

[Caruso *et al.*, 2024] Simone Caruso, Carmine Dodaro, Marco Maratea, Marco Mochi, and Francesco Riccio. CNL2ASP: converting controlled natural language sentences into ASP. *Theory Pract. Log. Program.*, 24(2):196–226, 2024.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, and Heewoo Jun et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[Dakhel *et al.*, 2023] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. Github copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734, 2023.

[Dubey *et al.*, 2019] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *ISWC (2)*, volume 11779 of *LNCS*, pages 69–78. Springer, 2019.

[Eiter *et al.*, 2007] Thomas Eiter, Michael Fink, and Stefan Woltran. Semantical characterizations and complexity of equivalences in answer set programming. *ACM Trans. Comput. Log.*, 8(3):17, 2007.

[Erdem and Yeniterzi, 2009] Esra Erdem and Reyyan Yeniterzi. Transforming controlled natural language biomedical queries into answer set programs. In *BioNLP@HLT-NAACL*, pages 117–124. ACL, 2009.

[Erdem *et al.*, 2016] Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of answer set programming. *AI Mag.*, 37(3):53–68, 2016.

[Ernst and Bavota, 2022] Neil A. Ernst and Gabriele Bavota. Ai-driven development is here: Should you worry? *IEEE Softw.*, 39(2):106–110, 2022.

[Fang and Tompits, 2017] Min Fang and Hans Tompits. An approach for representing answer sets in natural language. In *DECLARE*, volume 10997 of *LNCS*, pages 115–131. Springer, 2017.

[Febbraro *et al.*, 2011] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: integrated development environment for answer set programming. In *LPNMR*, volume 6645 of *LNCS*, pages 317–330. Springer, 2011.

[Gebser *et al.*, 2012] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[Gebser *et al.*, 2016] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASIcs*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[Gebser *et al.*, 2020] Martin Gebser, Marco Maratea, and Francesco Ricca. The seventh answer set programming competition: Design and results. *Theory Pract. Log. Program.*, 20(2):176–204, 2020.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Gener. Comput.*, 9(3/4):365–386, 1991.

[Hahn *et al.*, 2023] Susana Hahn, Orkunt Sabuncu, Torsten Schaub, and Tobias Stolzmann. Clingraph: A system for asp-based visualization. *CoRR*, abs/2303.10118, 2023.

[Hua and Wang, 2020] Xinyu Hua and Lu Wang. PAIR: planning and iterative refinement in pre-trained transformers for long text generation. *CoRR*, abs/2010.02301, 2020.

[Ishay *et al.*, 2023] Adam Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. In *KR*, pages 374–383, 2023.

[Jurafsky and Martin, 2009] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

[Kalliamvakou, 2022] Eirini Kalliamvakou. Research: quantifying github copilot's impact on developer productivity and happiness. *The GitHub Blog*, 2022.

[Kuhn, 2014] Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguistics*, 40(1):121–170, 2014.

[Lewis *et al.*, 2019] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[Lierler *et al.*, 2016] Yuliya Lierler, Marco Maratea, and Francesco Ricca. Systems, engineering environments, and competitions. *AI Mag.*, 37(3):45–52, 2016.

[Lifschitz, 2019] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[Lifschitz, 2022] Vladimir Lifschitz. Translating definitions into the language of logic programming: A case study. In *ICLP Workshops*, volume 3193 of *CEUR WP*. CEUR-WS.org, 2022.

[Mitra and Baral, 2016] Arindam Mitra and Chitta Baral. Addressing a question answering challenge by combining statistical methods with inductive rule learning and reasoning. In *AAAI*, pages 2779–2785. AAAI Press, 2016.

[Moldovan *et al.*, 2003] Dan I. Moldovan, Christine Clark, Sanda M. Harabagiu, and Steven J. Maiorano. COGEX: A logic prover for question answering. In *HLT-NAACL*. The Association for Computational Linguistics, 2003.

[Nye *et al.*, 2021] Maxwell I. Nye, Michael Henry Tessler, Joshua B. Tenenbaum, and Brenden M. Lake. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In *NeurIPS*, pages 25192–25204, 2021.

[Papineni *et al.*, 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318. ACL, 2002.

[Peng *et al.*, 2023] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from github copilot. *CoRR*, abs/2302.06590, 2023.

[Perevalov *et al.*, 2022] Aleksandr Perevalov, Dennis Diefenbach, Ricardo Usbeck, and Andreas Both. Qald-9-plus: A multilingual dataset for question answering over dbpedia and wikidata translated by native speakers. In *ICSC*, pages 229–234. IEEE, 2022.

[Raffel *et al.*, 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[Schwitter, 2018] Rolf Schwitter. Specifying and verbalising answer set programs in controlled natural language. *Theory Pract. Log. Program.*, 18(3-4):691–705, 2018.

[Stahlberg, 2020] Felix Stahlberg. Neural machine translation: A review. *J. Artif. Intell. Res.*, 69:343–418, 2020.

[Sutskever *et al.*, 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[Trivedi *et al.*, 2017] Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. Lc-quad: A corpus for complex question answering over knowledge graphs. In *ISWC (2)*, volume 10588 of *LNCS*, pages 210–218. Springer, 2017.

[Vos *et al.*, 2012] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating answer-set programs in lana. *Theory Pract. Log. Program.*, 12(4-5):619–637, 2012.

[Wagner, 2010] Wiebke Wagner. Steven bird, ewan klein and edward loper: Natural language processing with python, analyzing text with the natural language toolkit - o'reilly media, beijing, 2009, ISBN 978-0-596-51649-9. *Lang. Resour. Evaluation*, 44(4):421–424, 2010.

[Xu *et al.*, 2018] Jingjing Xu, Xuancheng Ren, Yi Zhang, Qi Zeng, Xiaoyan Cai, and Xu Sun. A skeleton-based model for promoting coherence among sentences in narrative story generation. In *EMNLP*, pages 4306–4315. ACL, 2018.

[Yang *et al.*, 2020] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. A survey of deep learning techniques for neural machine translation. *CoRR*, abs/2002.07526, 2020.

[Yang *et al.*, 2023] Zhun Yang, Adam Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text. In *ACL (Findings)*, pages 5186–5219. ACL, 2023.