

A Tool for Encoding Controlled Natural Language Specifications as ASP Rules

Carmine Dodaro¹, Marco Maratea² and Francesco Riccio¹

¹DeMaCS, University of Calabria, Italy

²DIBRIS, University of Genoa, Italy

Abstract

Answer Set Programming (ASP) is a popular declarative programming language for solving hard combinatorial problems. Albeit ASP has been widely adopted in both academic and industrial contexts, it might be difficult for people who are not familiar with logic programming conventions to use it. In this paper, we propose a translation of English sentences expressed in a controlled natural language (CNL) form into ASP. In particular, we first provide a definition of the type of sentences allowed by our CNL and their translation as ASP rules, and then exemplify the usage of CNL for the specification of well-known combinatorial problems.

Keywords

Answer set programming, Logic Programming, Controlled Natural Language

1. Introduction


Answer Set Programming (ASP) [1] is a well-known declarative programming paradigm geared toward solving hard combinatorial problems. As a matter of fact, ASP has been widely used for solving problems in both academic and industrial contexts [2]. The success of ASP is mainly due to its simple syntax and intuitive semantics, and to the availability of efficient systems [3, 4]. However, despite its success, the usage of ASP is still problematic for operators without mathematical or logical background. In this paper, we propose a system that automatically translates English sentences expressed in a controlled natural language (CNL) [5] into ASP rules. Such a CNL is oriented towards three different types of use cases, i.e., (i) offering a tool to non-ASP experts for specifying ASP programs; (ii) offering a fast prototyping tool to ASP experts for creating intuitive ASP encodings which are subsequently subject to optimization; and (iii) improving the readability of ASP programs, since there is a one-to-one mapping between ASP rules and English specifications. In particular, the CNL presented in this paper is inspired by the Semantics of Business Vocabulary and Business Rules (SBVR) [6, 7], which is a standard proposed by the Object Management Group to formally describe complex entities, e.g. the ones related to a business, using natural language. Previous attempts of translating CNL into ASP were proposed in [8, 9], and more recently, by Schwitter in [10] who defined the language PENG^{ASP}. Albeit

Datalog 2.0 2022: 4th International Workshop on the Resurgence of Datalog in Academia and Industry, September 05, 2022, Genova - Nervi, Italy

✉ carmine.dodaro@unical.it (C. Dodaro); marco.maratea@unige.it (M. Maratea); riccio.francesco@outlook.it (F. Riccio)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

some aspects of the grammar rules of PENG^{ASP} are also present in the grammar of our CNL, we propose a language that is oriented in the definition of combinatorial problems in a natural-feeling, but that is also reliable in its translation to ASP, avoiding using more than one grammar construct for the same behaviour and choosing words with an easily deducible meaning from the given context. Moreover, it should be noted that the implementation of PENG^{ASP} , as well as a binary executable, is not yet public, whereas the implementation of the system subject of this paper is open source and publicly-available at <https://github.com/dodaro/cnl2asp>.

2. Preliminaries

Answer Set Programming (ASP) [1] is a programming paradigm developed in the field of non-monotonic reasoning and logic programming. In this section, we overview the language of ASP [11].

Syntax. The syntax of ASP is similar to the one of Prolog. Variables are strings starting with an uppercase letter, and constants are non-negative integers or strings starting with lowercase letters. A *term* is either a variable or a constant. A *standard atom* is an expression $p(t_1, \dots, t_n)$, where p is a *predicate* of arity n and t_1, \dots, t_n are terms. An atom $p(t_1, \dots, t_n)$ is ground if t_1, \dots, t_n are constants. A *ground set* is a set of pairs of the form $\langle \text{consts} : \text{conj} \rangle$, where *consts* is a list of constants and *conj* is a conjunction of ground standard atoms. A *symbolic set* is a set specified syntactically as $\{\text{Terms}_1 : \text{Conj}_1; \dots, \text{Terms}_t : \text{Conj}_t\}$ where $t > 0$, and for all $i \in [1, t]$, each Terms_i is a non-empty list of terms, and each Conj_i is a non-empty conjunction of standard atoms. A *set term* is either a symbolic set or a ground set. Intuitively, a set term $\{X : a(X, c), p(X); Y : b(Y, m)\}$ stands for the union of two sets: the first one contains the X -values making the conjunction $a(X, c), p(X)$ true, and the second one contains the Y -values making the conjunction $b(Y, m)$ true. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Basically, aggregate functions map multisets of constants to a constant. The most common functions implemented in ASP systems are `#count`, for counting number of terms; and `#sum`, for computing sum of integers [12]. An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, <=, >, >=, !=, =\}$ is a comparison operator, and T is a term called guard. An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set. An *atom* is either a standard atom or an aggregate atom. A *rule* r has the following form:

$$a_1 \mid \dots \mid a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m.$$

where a_1, \dots, a_n are standard atoms (with $n \geq 0$), b_1, \dots, b_k are atoms, and b_{k+1}, \dots, b_m are standard atoms (with $m \geq k \geq 0$). A literal is either a standard atom a or its negation `not` a . The disjunction $a_1 \mid \dots \mid a_n$ is the *head* of r , while the conjunction $b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m$ is its *body*. Rules with empty body are called *facts*. Rules with empty head are called *constraints*. A variable that appears uniquely in set terms of a rule r is said to be *local* in r , otherwise it is a *global* variable of r . An ASP program is a set of *safe* rules, where a rule r is *safe* if the following conditions hold: (i) for each global variable X of r there is a positive standard atom ℓ in the body of r such that X appears in ℓ ; and (ii) each local variable of r appearing in a

symbolic set $\{\text{Terms} : \text{Conj}\}$ also appears in a positive atom in Conj . A *weak constraint* ω [13] is of the form:

$$:\sim b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m. [w@l, t_1, \dots, t_z]$$

where t_1, \dots, t_z are terms, w and l are the weight and level of ω , respectively. Intuitively, $[w@l]$ is read “as weight w at level l ”, where weight is the “cost” of violating the condition in the body of w , whereas levels can be specified for defining a priority among preference criteria. An ASP program with weak constraints is $\Pi = \langle P, W \rangle$, where P is a program and W is a set of weak constraints. A standard atom, a literal, a rule, a program or a weak constraint is *ground* if no variables appear in it.

Semantics. Let P be an ASP program. The *Herbrand universe* U_P and the *Herbrand base* B_P of P are defined as usual. The ground instantiation G_P of P is the set of all the ground instances of rules of P that can be obtained by substituting variables with constants from U_P . An *interpretation* I for P is a subset I of B_P . A ground standard atom p is true w.r.t. I if $p \in I$, and false otherwise. A literal $\text{not } p$ is true w.r.t. I if p is false w.r.t. I , and false otherwise. An aggregate atom is true w.r.t. I if the evaluation of its aggregate function (i.e., the result of the application of f on the multiset S) with respect to I satisfies the guard; otherwise, it is false. A ground rule r is *satisfied* by I if at least one atom in the head is true w.r.t. I whenever all conjuncts of the body of r are true w.r.t. I . A model is an interpretation that satisfies all rules of a program. Given a ground program G_P and an interpretation I , the *reduct* of G_P w.r.t. I is the subset G_P^I of G_P obtained by deleting from G_P the rules in which a body literal is false w.r.t. I [12]. An interpretation I for P is an *answer set* (or *stable model*) for P if I is a minimal model (under subset inclusion) of G_P^I (i.e., I is a minimal model for G_P^I). Given a program with weak constraints $\Pi = \langle P, W \rangle$ and an interpretation I , the semantics of Π extends from the basic case defined above. Thus, let G_P and G_W be the instantiation of P and W , respectively. Then, let G_W^I be the set

$$\{(w@l, t_1, \dots, t_z) \mid :\sim b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m. [w@l, t_1, \dots, t_z] \in G_W \text{ and } b_1, \dots, b_m \in I\}.$$

Moreover, for an integer l , $P_l^I = \sum_{(w@l, t_1, \dots, t_z) \in G_W^I} w$ if there is at least one tuple in G_W^I whose level is equal to l , 0 otherwise. Given a program with weak constraints $\Pi = \langle P, W \rangle$, an answer set M for P is said to be dominated by an answer set M' for P , if there exists an integer l such that $P_l^{M'} < P_l^M$ and $P_{l'}^{M'} = P_{l'}^M$ for all integers $l' > l$. An answer set M for P is said to be *optimal* or *optimum* for Π if there is no other answer set M' that dominates M [11].

Syntactic shortcuts. In the following, $p(1..n)$ denotes the set of facts $p(1) \dots p(n)$. Moreover, we use *choice rules* of the form $\{x\}$, where x is a set of atoms. Choice rules of this kind can be viewed as a syntactic shortcut for the rule $p \mid p'$ for each $p \in x$, where p' is a fresh new atom not appearing elsewhere in the program, meaning that the atom p can be chosen as true. Choice rules can also have bounds, i.e., $\{x\} = 1$, can be seen as a shortcut for the choice rule $\{x\}$ and the rule $:- \#count\{x\} != 1$.

3. CNL2ASP

This section deals with the implementation of the grammar for the Controlled Natural Language (CNL) presented in this work. A specification written in this CNL is made of propositions, the structure of which is defined by **clauses**, linked by connectives, that are used to express *concepts*, query them for information or express conditions on them. Concepts in a proposition can be subjects, that represent the focus of the definition and only one of them is allowed per proposition, or companion objects, that represent other information associated with the subject of the proposition.

The combination of clauses that produces a proposition defines its type, that is used to understand what the proposition is supposed to mean and how that meaning can be translated into ASP rules and facts.

In the following subsections, we show the proposition types that compose the CNL and related sub-types, if any, along with a visualization of their translation into the elements that form the resulting ASP programs.

3.1. Grammar

A brief description of the grammar is reported in the following, we refer the reader to [14] for the full grammar. In particular, our CNL is made of one or many propositions:

```

1 start → (( negative_strong_constraint_proposition
2         | positive_strong_constraint_proposition
3         | weak_constraint_proposition
4         | definition_proposition
5         | quantified_choice_proposition)CNL_END_OF_LINE)+

```

where each proposition is terminated by `CNL_END_OF_LINE` (in our case, a dot). We consider five types of propositions, namely negative strong constraints, positive strong constraints, weak constraints, definition clauses, and quantified choice clauses.

Negative strong constraints are as follows:

```

1 negative_strong_constraint_proposition → "it is prohibited that" (
   simple_clause ("and also" simple_clause)* (where_clause)? |
   aggregate_clause comparison_clause (where_clause)? |
   quantified_constraint (where_clause)?)

```

In particular, they start with the sentence `"it is prohibited that"` and are always followed by a simple clause, i.e. a sentence of the form `subject verb object`, by an aggregate clause, i.e. a sentence expressing a form of aggregations (e.g. `the number of`) or by a quantified constraint, i.e. a sentence used to specify clauses with quantifiers as `every` or `any`. After that, additional sentences can be added, which can be of the type *(i)* `where_clause`; or *(ii)* `comparison_clause`. Sentences of type *(i)* are used to specify conditions; sentences of type *(ii)* are used to specify comparison between elements (e.g. `X is equal to 1`).

Positive strong constraints are as follows:

```

1 positive_strong_constraint_proposition → "it is required that" (
   aggregate_clause comparison_clause (where_clause)? | when_then_clause (
   where_clause)? | quantified_constraint (where_clause)?)

```

In particular, they start with the sentence "it is required that" and are followed by an aggregate clause, by a quantified constraint, or by a clause of the type "when"/"then".

Weak constraints are as follows:

```
1 weak_constraint_proposition → "it is preferred that" ", "?
   weak_priority_operator ", "? "that" (condition_operation |
   aggregate_clause) weak_optimization_operator (where_clause)?
```

In particular, they start with the sentence "it is preferred that" and are always followed by a priority operator, i.e. a sentence expressing the level of relevance of the constraint with respect to other weak constraints (e.g. "with low priority") and either an aggregate clause or a condition operation, i.e., a sentence expressing operations between variables in the proposition (e.g. the sum of X and Y). The proposition is closed with an optimization operator, i.e., a sentence expressing the nature of the optimization (e.g. "is maximized") and an optional where clause.

Definition propositions are as follows:

```
1 definition_proposition → (compounded_clause | constant_definition_clause |
   enumerative_definition_clause)
```

In particular, a definition proposition can be one of the type (i) *compounded_clause*; (ii) *constant_definition_clause*; (iii) *enumerative_definition_clause*. Sentences of type (i) are used to define elements using lists and ranges; sentences of type (ii) are used to specify constants; sentences of type (iii) are used to define elements one at a time, optionally closing the proposition with a "when" clause, defining a condition in which the element is defined (e.g. X is true when Y is true), and with a where clause.

Quantified choice propositions are as follows:

```
1 quantified_choice_proposition → quantifier subject_clause "can" CNL_COPULA?
   (verb_name | verb_name_with_preposition) (
   quantified_exact_quantity_clause | quantified_range_clause)?
   quantified_object_clause? foreach_clause?
```

In particular, they start with a quantifier, and are always followed by a subject and a verb, optionally connected by a CNL_COPULA (e.g. is, is a, is an, ...) and then, also optionally, a sentence of either type (i) *quantified_exact_quantity_clause*; or (ii) *quantified_range_clause*. Sentences of type (i) are used to express the quantity in question in exact terms (e.g. exactly 1); sentences of type (ii) are used to express it using a range (e.g. between 1 and 2). The proposition can be closed with an object clause, i.e. a sentence used to express an object for the proposition, in a subject verb object fashion, or a simple clause and, finally, a *foreach clause*, i.e. a sentence used to express additional objects for which any possible value can be tried.

3.2. Definition propositions

Definition propositions are used to define concepts that can, then, be used by other propositions. These definitions express the *signature* of the concept indicated by the subject of the proposition, carrying information regarding the concept in the definition that our system can use later on in the specification whenever the same concept is used.

There are three different types of constructions available for these kinds of propositions, namely constant definitions, compound definitions, and enumerative definitions.

Constant definitions. They are used to introduce constants to be used later on in the specification.

Example 3.1.

MyConstant **is equal to** 10. (1)

As we can see from proposition (1), the constant 10 is introduced with a **equal to** clause, and it is bound to the subject of the proposition. In the case of constant definitions, there are no translations to ASP available, because they are instead stored and substituted in the resulting program when the subject of the definition is used.

Compound definitions. Such definitions are used to introduce a set of related concepts all at once, by making use of either ranges of numbers or lists.

Example 3.2.

A *day* **goes from** 1 to 365 **and is made of** hours that are made of minutes. (2)

A *drink* **is one of** alcoholic, nonalcoholic **and has color** that is equal to **respectively** blue, yellow. (3)

Proposition (2) is an example of a definition of a day using a range, identified by the construct **goes from/to**. Additionally, it can be noticed the presence of additional **made of** clauses, indicating what the preceding object, may it be the subject or an object complement, is made of.

Proposition (3) tries to do the same thing with a drink using lists with a **one of** clause, where one can also specify additional attributes for each element of the list in a positional way using a **respectively** clause, and a list with the same number of elements of the list enumerating all the possible values that the subject of the proposition can have.

The corresponding ASP code, in this case, is quite straightforward and is depicted below:

```
1 day(1..365).
2 drink(1, "alcoholic", "blue").
3 drink(2, "nonalcoholic", "yellow").
```

where line 1 corresponds to proposition (2), while lines 2 and 3 are related to proposition (3). One can notice the absence of information regarding **made of** clauses in proposition (2), but this is because the system only adds the composition hierarchy to the signature and does not encode them into ASP. Also, list elements defined in proposition (3) carry around their position number with them, which turns out to be handy as a basic way to encode precedence relationships when the subject is not a number.

Enumerative definitions. Enumerative definitions are used to introduce a property for a single concept or a relationship between a set of concepts. The peculiarity of this kind of propositions lies in the different translations into ASP as the clauses used within them change.

Example 3.3.

John is a waiter. (4)

Pub 1 is close to pub 2 and pub X, where X is equal to 3,4. (5)

Waiter John works in pub 1. (6)

Waiter W is working when waiter W serves a drink. (7)

In propositions (4)–(7) the construction to define relationships or properties related to a particular subject is shown. Proposition (5), in particular, illustrates another feature of our CNL, i.e., **where** clauses, that are used in the example to define the values that the variable *X* can take.

The previous definitions always hold true, whereas proposition (4) is a **conditional** definition, identified by a **when** clause, and so it holds true only if the statement introduced by the aforementioned clause is true. Note that John is not considered as a variable here since it contains a lower case letter. Indeed, our assumption is that variables contain only upper case letters, numbers or symbols.

Below are the translations in ASP of these examples:

```
1 waiter(john).
2 close_to(1, 2, 3). close_to(1, 2, 4).
3 work_in(john, 1).
4 working(W) :- serve(W,_).
```

Also here, the translation is quite intuitive. Propositions that express relationships or properties that always hold true become regular ASP facts. In particular, proposition (6) is translated in a similar manner to compound definitions with lists. Proposition (5), instead, represents something that follows by the truth of another statement and, as such, is translated into an ASP rule, with what's inside the **when** clause as the body of the rule.

3.3. Quantified propositions

Quantified propositions are used to define relationships or properties that *can* be true for a given set of selected concepts following a choice. Also these propositions define a *signature* for the concept upon which the choice has to be made. Quantified propositions are always introduced by the **every** quantifier and, since they express possibilities, always contain a **can** clause.

Example 3.4.

Every patron can drink in exactly 1 pub for each day. (8)

Every waiter can serve a drink. (9)

Every patron can drink with a patron next to patron X. (10)

Proposition (8) shows how one can express an exact number of choices that can be made for the concept expressed by the subject, and also how other concepts can be used in tandem with the subject to create a sort of cartesian product of choices, using a **for each** clause. These last constructions are optional, as shown in proposition (9). Proposition (10), instead, shows how one can constrain the choices that can be made by means of an auxiliary statement that defines the set of concepts from which the set of possible choices can be constructed. Their full translations into ASP is shown below:

```
{drink_in(_X1, _X2, _X3): pub(_X2)} = 1 :- patron(_X1), day(_X3).
{serve(_X1, _X2): drink(_X2)} :- waiter(_X1).
{drink_with(X, _X1): next_to(X, _X1)} :- patron(X).
```

These translations present a couple of novelties. The first one is the introduction of choice rules with bounds, that are the ASP constructs that make it possible to represent propositions of this type. The second one is the introduction of generated variables (starting with symbol `_`) that are used wherever two atoms have to be bound and no variable to use has been found in the specification given in input. This feature enables the specification writer to avoid cluttering the document with unnecessary variables, as can be seen throughout the propositions, with the only limitation that anaphoras have to be expressed explicitly by providing the correct variable.

3.4. Strong Constraint propositions

Strong Constraint propositions are used to define assertions that *must* be true for a given set of selected concepts. This kind of propositions does not introduce new *signatures* but, on the contrary, it consumes other signatures that were previously defined, meaning that the concepts used inside these are to be defined *before* they are used. A strong constraint can represent either a prohibition (**It is prohibited...**) or a requirement (**It is required...**), in which case one must specify a **when** clause, containing a premise, and a **then** clause, containing a consequence.

In addition to **simple** clauses, that are made of subject, a verb, and related object clauses, one can also specify **aggregate** clauses, either in *active* or *passive* form, that define an aggregation of the set of concepts that satisfy the statement in their body with the operator that was specified (**number** or **total**).

Example 3.5.

It is prohibited that *waiter X works with waiter W* and also *waiter X works with a waiter before W*, **where** *W is between Jack and Mary*. (11)

It is required that the number of occurrences between each 5 days where a waiter works is at least 2. (12)

It is required that when a waiter works **for 2 consecutive days** then the next day does not work. (13)

It is prohibited that a waiter does not work in a day and also the previous 2 consecutive days does not work. (14)

It is required that when waiter X works in pub P1 **then** waiter X does not work in pub P2, **where** P1 is different from P2. (15)

It is required that every waiter is payed. (16)

Proposition (11)-(14) show practical examples of how lists can be queried for sequence relationships between its elements. It is possible to check whether elements come before or after a specified element, take "windows" of a fixed number of elements or take consecutive elements, also from a specified one. Proposition (11) and proposition (15) show the other two features supported by **where** clauses, making it possible to express bounds or equality constraints between variables. In addition to that, arithmetic operations are supported by the use of an **equal to** clause. Lastly, proposition (16) is an example of how to specify a requirement that must hold for all the elements present in a particular set of concepts. The translations of the

propositions above are the following:

```

1 :- waiter(_X1, W), waiter(_X2, _X3), work_with(X, W), work_with(X, _X3), _X3
   < W, _X1 >= 1, _X1 <= 3, _X2 >= 1, _X2 <= 5.
2 :- waiter(_X1), day(_X2), _X2 <= 352, #count{_X3: work(_X1, _X3), _X3 >=
   _X2, _X3 <= _X2+4} < 2.
3 :- day(_X1), #count{_X3: work(_X2, _X3), _X3 >= _X1, _X3 <= _X1+1} = 2, not
   work(_X2, _X1+2), waiter(_X2).
4 :- work(_X1, _X2), day(_X3), #count{_X2: work(_X1, _X2), _X2 >= _X3-2, _X2 <=
   _X3-1} = 2.
5 :- work_in(_X1, P1), work_in(_X1, P2), P1 != P2.
6 :- not payed(_X1), waiter(_X1).

```

The translations above show that list element operations and checks are reflected on the indices of the elements in the list, or the numbers directly when there is a list. This behaviour is shown in the translation from proposition (11) and proposition (12). Proposition (13) and proposition (14), instead, show how check for consecutive elements in list are translated, as an existence check of the exact number of elements that have to satisfy the statement inside the aggregate, where the actual statement expressed in the proposition is expressed. On the other hand, proposition (15) and proposition (16) translations are not very complex, and show how different clauses referring to the same or different subject are linked in the resulting program, as well as how quantifiers inside constraints are dealt with.

3.5. Weak Constraint propositions

Weak constraint propositions are used to define assertions that are *preferably* true for a given set of selected concepts. Also this type of propositions consume *signatures* from previously defined concepts. They are always introduced by **It is preferred** and need the specification of the optimization objective (either **minimization** or **maximization**) and the level of priority of the optimization (**low**, **medium** or **high**).

Example 3.6.

It is preferred, with high priority, that the difference in absolute value between 1000 and and the total of items in a box where a drink is stocked ranging between 10 and 5000 is minimized. (17)

It is preferred with low priority that the number of drinks that are served is maximized. (18)

Proposition (17) displays a weak constraint that represent a minimization objective. Here, the value to optimize is obtained by means of an arithmetic operation (showing here that also the absolute value operator for those is supported) between a constant number and the result of an aggregate clause, also with the definition of the range of possible values where the result of this arithmetic operation lies (useful for limiting the search space of the solutions). In proposition (18) there is an example of a maximization constraint, showing that arithmetic operations are optional and even just the results of the aggregate clauses can be used as values to be optimized.

Translation of the propositions above are shown below:

```

1 :~ drink(_X1), #sum{_X2, _X3: stocked(_X1, _X3, _X2)} = _X4, _X4 >= 10, _X4 <=
   5000, _X5 = |1000-_X4|. [_X5@1, _X1]

```

```
2 :~ #count{ _X1: served(_X1)} = _X2. [-_X2@3]
```

The translations of proposition (17) and proposition (18) are very linear, starting from the aggregate for the first value, and then, if necessary, putting bounds on it or extracting another result, that is, the one to be optimized. Remarkably, priority levels are transformed into numerical level constants, in descending order from the least severe to the most, and maximization constraints are just translated as flipping the sign of the value to be optimized.

Lastly, it is worth mentioning that the difference in the translation of the aggregate from proposition (18) with respect to other translations seen in previous examples is due to the fact that this is a special case of the *active* aggregate form, where a property is in verb position.

4. Use cases

In this section we present some examples to demonstrate how the language can be used to define well-known problems in a natural and easily understandable way. The corresponding translations into ASP are also provided.

4.1. Graph coloring

We begin by presenting an encoding of the graph coloring problem using our controlled natural language:

```
1 A node goes from 1 to 3.
2 A color is one of red, green, blue.
3 Node 1 is connected to node X, where X is one of 2, 3.
4 Node 2 is connected to node X, where X is one of 1, 3.
5 Node 3 is connected to node X, where X is one of 1, 2.
6 Every node can be assigned to exactly 1 color.
7 It is required that when node X is connected to node Y then node X is not
  assigned to color C and also node Y is not assigned to color C.
```

One can notice the presence of a ranged definition proposition (line 1) and a list definition proposition (line 2), enumerative definition propositions with where clauses (lines 3–5), a quantified clause (line 6) and, lastly, a *positive strong constraint* (line 7). The resulting ASP encoding is the following:

```
1 node(1..3).
2 color(1,"red"). color(2,"green"). color(3,"blue").
3 connected_to(1,2). connected_to(1,3).
4 connected_to(2,1). connected_to(2,3).
5 connected_to(3,1). connected_to(3,2).
6 {assigned_to(_X1,_X2): color(_, _X2)} = 1 :- node(_X1).
7 :- connected_to(X,Y), assigned_to(X,C), assigned_to(Y,C).
```

where each proposition at line i is translated as the rule(s) reported at line i (with $i = [1..7]$).

For the sake of completeness, we report a comparison with the CNL for specifying the graph coloring problem used by PENG^{ASP} (Figure 5 of [10]).

```
1 The node 1 is connected to the nodes 2 and 3.
```

- 2 The node 2 is connected to the nodes 1 and 3.
- 3 The node 3 is connected to the nodes 1 and 2.
- 4 Red is a colour. Green is a colour. Blue is a colour.
- 5 Every node is assigned to exactly one colour.
- 6 It is not the case that a node X is assigned to a colour and a node Y is assigned to the colour and the node X is connected to the node Y.

There are two main differences. The first one is that our CNL must use variables (i.e., x in our example) also to specify the connections, whereas the one of PENG^{ASP} does not need it. In our case, sentence at line 1 would create the atom `connected_to(1,2,3)`. Secondly, the last sentence is expressed in a negative form in case of PENG^{ASP} , which is similar to the concept of constraint in ASP, whereas our CNL uses a positive sentence which is similar to the concept of clause in propositional logic.

4.2. Hamiltonian path

The following statements represent an encoding in our CNL for the Hamiltonian path problem:

- 1 A node goes from 1 to 5.
- 2 Node 1 is connected to node X, where X is one of 2, 3.
- 3 Node 2 is connected to node X, where X is one of 1, 4.
- 4 Node 3 is connected to node X, where X is one of 1, 4.
- 5 Node 4 is connected to node X, where X is one of 3, 5.
- 6 Node 5 is connected to node X, where X is one of 3, 4.
- 7 Every node X can have a path to a node connected to node X.
- 8 It is required that the number of nodes where node X has a path to is equal to 1.
- 9 It is required that the number of nodes that have a path to node X is equal to 1.
- 10 Node Y is reachable when node X is reachable and also node X has a path to node Y.
- 11 It is required that every node is reachable.
- 12 Start is equal to 1.
- 13 Node start is reachable.

Line 1 defines the nodes and lines from 2 to 6 define the connections between nodes. Then, line 7 reports a quantified proposition with an object accompanied by a verb clause, lines 8 and 9 report strong constraint propositions with aggregates, line 10 reports a conditional definition clause, line 11 reports a constraint clause with the presence of a quantifier, and line 12 define the constant `start`, which is subsequently used in line 13. The ASP encoding corresponding to the CNL statements is the following:

```

1 node(1..5).
2 connected_to(1,2). connected_to(1,3).
3 connected_to(2,1). connected_to(2,4).
4 connected_to(3,1). connected_to(3,4).
5 connected_to(4,3). connected_to(4,5).
6 connected_to(5,3). connected_to(5,4).
7 {path_to(X,_X1): connected_to(X,_X1)} :- node(X).
8 :- node(X), #count{ _X2: path_to(X,_X2)} != 1.
```

```

9 :- node(X), #count{ _X3: path_to(_X3,X) } != 1.
10 reachable(Y) :- reachable(X), path_to(X,Y).
11 :- not reachable(_X4), node(_X4).
12 reachable(1).

```

where a CNL statement at line i is represented by the rule(s) at line i with ($i = [1..11]$), whereas CNL statements reported in lines 12 and 13 are encoded by the rule at line 12.

4.3. Maximal clique

The following statements represent the encoding in our CNL of the maximum clique problem:

```

1 A node goes from 1 to 5.
2 Node 1 is connected to node X, where X is one of 2, 3.
3 Node 2 is connected to node X, where X is one of 1, 3, 4, 5.
4 Node 3 is connected to node X, where X is one of 1, 2, 4, 5.
5 Node 4 is connected to node X, where X is one of 2, 3, 5.
6 Node 5 is connected to node X, where X is one of 2, 3, 4.
7 Every node can be chosen.
8 It is required that when node X is not connected to node Y then node X is
   not chosen and also node Y is not chosen, where X is different from Y.
9 It is preferred with high priority that the number of nodes that are chosen
   is maximized.

```

where statements from line 1 to line 6 define the input graph. Then, line 7 reports a quantified proposition with no object, line 8 contains a strong constraint proposition with a comparison on the variables used inside it, and line 9 reports a weak constraint expressing a maximization preference on the highest priority level. The resulting ASP encoding is reported in the following:

```

1 node(1..5).
2 connected_to(1,2). connected_to(1,3).
3 connected_to(2,1). connected_to(2,3). connected_to(2,4). connected_to(2,5).
4 connected_to(3,1). connected_to(3,2). connected_to(3,4). connected_to(3,5).
5 connected_to(4,2). connected_to(4,3). connected_to(4,5).
6 connected_to(5,2). connected_to(5,3). connected_to(5,4).
7 {chosen(_X1)} :- node(_X1).
8 :- not connected_to(X,Y), chosen(X), chosen(Y), X != Y.
9 :~ #count{ _X1: chosen(_X1) } = _X2. [-_X2@1]

```

where each CNL proposition at line i is translated as the rule(s) reported at line i (with $i = [1..9]$).

5. Conclusion

In this paper we proposed a system for automatically translating English sentences expressed using a controlled language into ASP rules. Moreover, we provided several examples of combinatorial problems that can be specified in a clear and easy way, even by a non-ASP expert, using our CNL and their translation as ASP rules. Concerning future work, our CNL might be extended to cover additional constructs and language extensions, e.g. temporal operators [15]

or Constraint ASP (CASP) [16, 17], and the tool can be extended to implement a bidirectional conversion, where ASP rules are translated into CNL statements. Finally, we recall that the tool presented in this paper is open source and available at <https://github.com/dodaro/cnl2asp>.

References

- [1] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, *Commun. ACM* 54 (2011) 92–103. doi:10.1145/2043174.2043195.
- [2] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, *AI Mag.* 37 (2016) 53–68. doi:10.1609/aimag.v37i3.2678.
- [3] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: *Technical Communications of ICLP*, volume 52 of *OASICS*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:15. doi:10.4230/OASICS.ICLP.2016.2.
- [4] M. Alviano, F. Calimeri, C. Dodaro, D. Fuscà, N. Leone, S. Perri, F. Ricca, P. Veltri, J. Zangari, The ASP system DLV2, in: *Proceedings of LPNMR*, volume 10377, Springer, 2017, pp. 215–221. URL: https://doi.org/10.1007/978-3-319-61660-5_19. doi:10.1007/978-3-319-61660-5_19.
- [5] T. Kuhn, A survey and classification of controlled natural languages, *Comput. Linguistics* 40 (2014) 121–170. URL: https://doi.org/10.1162/COLI_a_00168. doi:10.1162/COLI_a_00168.
- [6] I. S. Bajwa, M. G. Lee, B. Bordbar, SBVR business rules generation from natural language specification, in: *AI for Business Agility, AAAI*, 2011. URL: <http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2378>.
- [7] The Business Rules Group, Defining business rules ~ what are they really?, 2000. URL: http://www.businessrulesgroup.org/first_paper/BRG-whatBR_3ed.pdf.
- [8] C. Baral, J. Dzifcak, Solving puzzles described in english by automated translation to answer set programming and learning how to do that translation, in: *Proceedings of KR*, AAAI Press, 2012.
- [9] E. Erdem, R. Yeniterzi, Transforming controlled natural language biomedical queries into answer set programs, in: *Proceedings of the BioNLP Workshop*, Association for Computational Linguistics, 2009, pp. 117–124.
- [10] R. Schwitter, Specifying and verbalising answer set programs in controlled natural language, *Theory Pract. Log. Program.* 18 (2018) 691–705. doi:10.1017/S1471068418000327.
- [11] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, *Theory Pract. Log. Program.* 20 (2020) 294–309. doi:10.1017/S1471068419000450.
- [12] W. Faber, G. Pfeifer, N. Leone, Semantics and complexity of recursive aggregates in answer set programming, *Artif. Intell.* 175 (2011) 278–298. doi:10.1016/j.artint.2010.04.002.
- [13] F. Buccafurri, N. Leone, P. Rullo, Enhancing disjunctive datalog by constraints, *IEEE Trans. Knowl. Data Eng.* 12 (2000) 845–860. doi:10.1109/69.877512.

- [14] C. Dodaro, M. Maratea, F. Riccio, Grammar of the cnl, 2022. URL: https://github.com/dodaro/cnl2asp/blob/main/src/cnl/grammar/cnl_grammar.lark.
- [15] P. Cabalar, M. Diéguez, T. Schaub, A. Schuhmann, Towards metric temporal answer set programming, *Theory Pract. Log. Program.* 20 (2020) 783–798. URL: <https://doi.org/10.1017/S1471068420000307>. doi:10.1017/S1471068420000307.
- [16] M. Balduccini, Industrial-size scheduling with ASP+CP, in: *Proceedings of LPNMR*, volume 6645 of *LNCS*, Springer, 2011, pp. 284–296. URL: https://doi.org/10.1007/978-3-642-20895-9_33. doi:10.1007/978-3-642-20895-9_33.
- [17] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, Clingcon: The next generation, *Theory Pract. Log. Program.* 17 (2017) 408–461. URL: <https://doi.org/10.1017/S1471068417000138>. doi:10.1017/S1471068417000138.