The Grammar of PENG^{ASP} Explained

Rolf Schwitter
Department of Computing
Macquarie University
Sydney, NSW, 2109, Australia
`Rolf.Schwitter@mq.edu.au`

Abstract

In this paper we present the controlled language and the grammar of the PENG^{ASP} system and explain how the new version of the grammar has been implemented in a logic programming framework. The grammar is now bi-directional and can be used to translate a specification written in controlled language into an executable answer set program and vice versa. The grammar is highly configurable for different application scenarios and can be used for incremental text processing together with a predictive authoring tool.

1 Introduction

A controlled language is a restricted version of a natural language which has been engineered by reducing the complexity of its grammar and/or its vocabulary to meet a particular purpose (Schwitter, 2010; Kittredge, 2003). Human-oriented controlled languages aim to improve the communication between humans or the readability of technical documentation for humans who are often not native speakers of the language. Machine-oriented controlled languages aim to improve machine translation of (technical) documentation or to support automatic reasoning via the translation of the language into a knowledge representation language.

PENG^{ASP} (Guy and Schwitter, 2017) is a machine-oriented controlled language and is in this respect similar to Attempto Controlled English (Fuchs et al., 2008), Computer-Processable Language (Clark et al., 2005), and Logical English (Kowalski, 2020). However, in contrast to these other three controlled languages, specifications written in PENG^{ASP} are exclusively translated into executable answer set programs (Gelfond and Kahl, 2014; Gelfond and Lifschitz, 1988). Answer set programming offers a rich declarative knowledge representation language for non-monotonic reasoning and is supported by high performance reasoning tools (Gebser et al., 2019). In contrast to the grammar of the PENG^{ASP} system introduced in Guy and Schwitter (2017), the latest version of the grammar is now bi-directional. This means that the same grammar can be used for processing a specification and for verbalising an answer set program.

According to the PENS scheme (Kuhn, 2014), controlled languages can be classified along four dimensions: precision (`P`), expressiveness (`E`), naturalness (`N`), and simplicity (`S`). Each of these dimensions is then measured on a scale of `1` to `5`. In addition to these four dimensions, nine properties are used to identify the type of a controlled language. Following this scheme, the controlled language PENG^{ASP} can be classified as $\mathtt{P}^5\mathtt{E}^3\mathtt{N}^4\mathtt{S}^3$ `A,W,F`. This means PENG^{ASP} is a language with fixed semantics ($\mathtt{P}^5$); offers medium expressive power ($\mathtt{E}^3$); uses natural sentences ($\mathtt{N}^4$); and requires a description of more than 10 pages ($\mathtt{S}^3$). Furthermore, PENG^{ASP} originated from academia (`A`), is intended to be written (`W`), and to be formally (`F`) represented as an answer set program. In other words, PENG^{ASP} is a high-level specification language for answer set programs that combines the readability and understandability of natural language with the precision and expressiveness of a declarative knowledge representation language.

The rest of this paper is structured as follows: In Section 2, we outline the requirements to the grammar of PENG^{ASP}. In Section 3, we give a brief introduction to answer set programming, since PENG^{ASP} is closely related to this formal target language. In Section 4, we introduce a motivating example that illustrates some features of the language and show how the corresponding answer set program looks like. In Section 5, we reveal more details about the design of the controlled language with a particular focus on the usage of certain grammatical constructions. In Section 6, we take a look at the implementation of the bi-directional grammar; and in Section 7, we conclude.

2 Requirements to PENG^{ASP}

The controlled language PENG^{ASP} and its grammar have been designed with a number of requirements in mind. Firstly, a controlled language specification should be translatable into an executable answer set program. Secondly, the grammar for this language should be highly configurable for different application scenarios, also for scenarios that do not necessarily require the full power of answer set programming. Thirdly, the same grammar should support the processing of a specification and the verbalisation of an answer set program; therefore, the grammar should be bi-directional. Fourthly, the grammar should also be useful to support the writing process of a specification in an incremental way, in particular with respect to generating lookahead information and resolving anaphoric expressions.

A possible processing strategy is to translate a controlled language specification into a syntax tree and then transform this tree into an answer set program. A better strategy is to generate the answer set program directly during the parsing process and design the grammar in such a way that it can serve as a language processor as well as a language generator. To achieve this, we start from a definite clause grammar and specify the grammar rules for the controlled language PENG^{ASP} in this unification-based notation (Pereira and Warren, 1980), and then use SWI Prolog (Wielemaker et al., 2012) as programming language. However, when Prolog is directly used to evaluate a definite clause grammar in a system like ours that heavily relies on incremental processing and user interaction, then Prolog's backtracking search strategy is not optimal, since it forgets all the previous work that it has done after a user interaction. To solve this problem, we use a chart parser in conjunction with the definite clause grammar to get the effect of a more complete parsing strategy that remembers substructures that it has already parsed (Gazdar and Mellish, 1989). For the processing of a specification, we transform the definite clause grammar into an alternative notation using a logic programming technique called term expansion (Wielemaker et al., 2012). The resulting notation is easier to process by a chart parser and the chart can then be used to extract the information that is required to support the writing process on the user interface level. For the verbalisation of an existing answer set program, the definite clause grammar can directly be used and does not need to be transformed into another format for chart parsing, since verbalisation does not require any user interaction in our system.

Bi-directionality is a key feature of our grammar and distinguishes the PENG^{ASP} system from other controlled language processors (Fuchs et al., 2008; Clark et al., 2005). Bi-directionality requires that we can (a) feed a specification *S* as input to the grammar *G* and get an answer set program *A* as output, and (b) feed the answer set program *A'* as input to the same grammar *G* and get a semantically equivalent version *S'* of the original specification as output. Figure 1 illustrates this form of lossless semantic round-tripping (Schwitter, 2020).

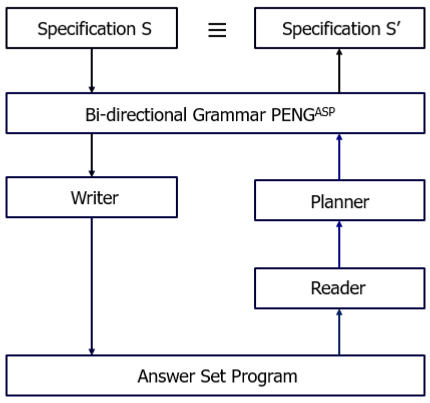

Figure 1: Round-tripping in PENG^{ASP}

In order to achieve semantic round-tripping, we use a Writer module that converts the internal version of the answer set program built by the grammar into an executable answer set program. In the case of verbalisation, a Reader module is used to read the executable answer set program and to produce a linguistically processable version of that answer set program. Since this processable version may contain certain redundancies, a Planner module is used that applies micro-planning tactics to aggregate redundant information such as subject aggregation and to deal with the identification of definite descriptions. The output of the Planner is a more compact version of the answer set program that is sent to the grammar and used to verbalise the answer set program.

3 Answer Set Programming

Answer set programming (ASP) is a declarative programming paradigm for knowledge representation and reasoning (Gelfond and Kahl, 2014; Gelfond and Lifschitz, 1988). ASP has been developed in the field of logic programming and nonmonotonic reasoning and has been applied to a wide range of areas in artificial intelligence (Erdem et al., 2016). ASP is supported by powerful reasoning tools and offers a rich representation language that allows for recursive definitions, negation, constraints, aggregates, optimization statements, and external functions (Gebser et al., 2019). An ASP program consists of a set of rules of the following form:

$\texttt{h}_1 \texttt{ ;...; } \texttt{h}_m \texttt{ :- } \texttt{b}_1 \texttt{ ,..., } \texttt{b}_n\texttt{.}$

Here each $\texttt{h}_i$ is a classical atom and $\texttt{b}_i$ is a literal for $\texttt{m} \geq \texttt{0}$ and $\texttt{n} \geq \texttt{0}$. A classical atom $\texttt{h}_i$ is either a positive atom of the form $\texttt{p(t}_1\texttt{,...,t}_k\texttt{)}$ or its strong negation of the form $\texttt{-p(t}_1\texttt{,...,t}_k\texttt{)}$, where `p` is a predicate name, $\texttt{t}_1\texttt{,...,t}_k$ are terms, and `k` $\geq \texttt{0}$ is the arity of the predicate name. A literal $\texttt{b}_i$ is of the form *A* or `not` *A*, where *A* is a classical atom or an atom over a built-in comparison predicate used to compare terms, and the connective `not` denotes weak negation (aka negation as failure or default negation). Note that a literal of the form `not` *A* is assumed to hold unless the atom *A* is derived to be true. In contrast, strong negation of an atom holds only if it can be derived. The if-connective '`:-`' separates the head of a rule from its body. Intuitively, if all positive literals in the body of a rule are true and all negative literals are satisfied, then the head of the rule must be true. The connective '`;`' denotes a disjunctive head. A disjunctive head holds if at least one of its atoms is true. An ASP rule with an empty body (and without the if-connective) is called a fact, and an ASP rule with an empty head (but with the if-connective) is called an integrity (strong) constraint.

An extension of ASP that is relevant for our work are choice rules. A choice rule has the form:

$\texttt{l\{e}_1 \texttt{ ;...; } \texttt{e}_m\texttt{\}u :- b}_1 \texttt{ ,..., } \texttt{b}_n\texttt{.}$

Here $\texttt{e}_i$ is a choice element of the form $\texttt{a:L}_1\texttt{,...,L}_k$, where `a` is a classical atom, $\texttt{L}_i$ are literals, and `l` and `u` are integers which express lower and upper bounds on the cardinality of elements. Intuitively, a choice rule means that if the body of the rule is true, then an arbitrary number of elements can be chosen as true as long as this number complies with the upper and lower bounds.

Note also that the input language to the ASP tool *clingo* (Gebser et al., 2019) supports double default negated literals in strong constraints. Furthermore, the language also supports weak constraints. In contrast to strong constraints, weak constraints do not eliminate answer sets but weight and prioritise them; more about this later.

4 A Motivating Example

The grammar of the PENG^{ASP} system translates a specification written in controlled language into an executable ASP program. The following example is an excerpt of a specification that contains information about students and their enrolments.

1. COMP3160 and COMP3220 are units.
2. Liam is a student and Olivia is a student.
3. Every student is either enrolled in COMP3160 or is enrolled in COMP3220.
4. If a student withdraws from a unit then the student is not enrolled in that unit.
5. It is not the case that Liam is enrolled in COMP3220.
6. Olivia withdraws from COMP3160.
7. Who is enrolled in COMP3220?

This specification consists of class assertions in (1) and (2); two conditional statements in (3) and (4); a constraint in (5), an unconditional statement in (6), and a wh-question in (7).

Listing 1: Answer Set Program

```
named(1, comp3160). class(1, unit).
named(2, comp3220). class(2, unit).
named(3, liam). class(3, student).
named(4, olivia). class(4, student).
1 { prop(X, 1, enrolled_in) ;
    prop(X, 2, enrolled_in) } 1  :-
  class(X, student).
-prop(X, Y, enrolled_in) :-
  class(X, student),
  pred(X, Y, withdraw_from),
  class(Y, unit).
:- prop(3, 2, enrolled_in).
pred(4, 1, withdraw_from).
answer(PN) :-
  named(X, PN), prop(X, 2, enrolled_in).
```

As we can see in Listing 1, the translation of the class assertions in (1) and (2) results in a number of facts in the ASP program. The two conditional statements are translated into two ASP rules. The first one (3) is translated into a choice rule that implements an exclusive disjunction describing alternative ways to form answer sets. The second one (4) is translated into a rule that contains a strongly negated atom as head. This rule eliminates answer set solutions if the body of the rule is true. The constraint in (5) results in a strong constraint in ASP and weeds out a particular solution from the generated answer sets. The statement in (6) is translated into a fact and the wh-question in (7) into an ASP rule with a specific atom (`answer/1`) in its head.

The ASP program uses a reified notation with a small number of predefined predicate atoms (e.g., `class/2`, `named/2`, `pred/3`, and `prop/3`). These predicate atoms can take variables, constants, positive numbers, or functional terms as arguments. Constants represent content words that occur in a specification and positive numbers replace existentially quantified variables in the program.

5 The Language: PENG^{ASP}

PENG^{ASP} can be used to make statements, enforce constraints, ask questions about a specification, and issue directives. There is not enough space in this section to cover all grammatical constructions of the language; therefore, we focus on the most important ones and provide selected examples. Syntactically, PENG^{ASP} distinguishes between simple and composite sentences. Each sentence consists of one or more clauses, and each clause has one main verb. Within a composite sentence, clauses may be joined via coordination or subordination, thus forming a compound or a complex sentence, respectively. The tense of the verb in a clause is either simple present, present continuous or future continuous depending on how the clause is used.

It is useful to introduce the concept of a core clause that forms the building block for simple and composite sentences in PENG^{ASP}. A core clause has the following canonical structure:

```
subject + predicator + (complements)
```

The subject and the predicator are always mandatory in a core clause. The subject is realised by a noun phrase and the predicator by a verb of a verb phrase. The selection of complements depends on the verb; dropping a complement either results in an incomplete core clause or a significant change in the meaning of the verb. A core clause forms the predicate-argument structure of a simple sentence. Adjuncts can follow the complement(s), but they are always optional, since they are not required to complete the meaning of a core clause. Phrases that occur in adjunct position serve exclusively as verbal modifiers; they add additional information like spatial or temporal information to the meaning of a core clause.

5.1 Making Statements

Simple statements can be made with the help of a simple sentence like (8) that is based on a core clause. The verb of this sentence can be modified for instance by a prepositional phrase that occurs in adjunct position; and this modification can be expressed as part of a simple sentence like (9). A positive clause like (9) can be rendered negative by the insertion of a strong negation (10).

8. Liam arrives.
9. Liam arrives at 09:00.
10. Liam does not arrive at 09:00.

Complex statements can be expressed with the help of compound sentences like (11), complex sentences like (12), verb phrase coordination like (13), and noun phrase coordination for class assertions like (14).

11. Liam studies at Macquarie University and Liam is enrolled in COMP3160.
12. Liam who studies at Macquarie University is enrolled in COMP3160.
13. Liam studies at Macquarie University and is enrolled in COMP3160.
14. Liam, Olivia, and Rona are students.

The compound sentence (11) uses two independent clauses that are joined by a coordinating conjunction (*and*). The complex sentence (12) consists of an independent clause and a dependent clause in the form of an embedded relative clause that modifies the proper name with the help of a subordinating conjunction (*who*). In (13) verb phrase coordination shares the same subject and in (14) noun phrase coordination (enumeration) shares the same complement. Note that the sentences (11-13) are syntactic variations of each other and result in the same ASP representation.

5.2 Making Conditional Statements

Like simple statements, simple conditional statements can also be expressed with a simple sentence that is based on a core clause. But in this case, the sentence requires a universally quantified noun phrase in subject position (15).

15. Every student is enrolled in at most 4 units.

Alternatively, a conditional sentence like (16) consisting of a dependent clause (expressing the condition) and a main clause (expressing the consequent) can be used to make the same statement.

16. If there is a student then the student is enrolled in at most 4 units.

Combining weak and strong negation in the same conditional sentence (17) allows us to express the closed-world assumption with respect to a given atom; meaning that all students who are not enrolled in a unit are explicitly known after processing the corresponding ASP rule.

17. If a student is not provably enrolled in a unit then the student is not enrolled in that unit.
18. If a person is holding an object at a time point and the person delivers that object at the same time point then the person will no longer be holding the object afterwards.

The conditional sentence (18) is interesting, since it describes an effect axiom (Mueller, 2015) for a temporal PENG^{ASP} specification. The condition contains a verb in present continuous tense that denotes a state and a verb in present tense that denotes an event. The consequent uses a verb in future continuous tense that describes the effect of the axiom.

5.3 Enforcing Constraints

In PENG^{ASP} constraints can be used to enforce conditions in a specification that must not become true. Syntactically, a constraint like (19) starts with a keyphrase (in brackets below for illustration purposes), followed by a potentially composite sentence.

19. [It is not the case that] a student who is enrolled in COMP3160 arrives at 11:00.

This composite sentence can have the same syntactic form as a sentence that can occur in the condition of a conditional sentence. In our case, the keyphrase is followed by a complex sentence that contains an embedded relative clause.

5.4 Asking Questions

PENG^{ASP} distinguishes between yes-no questions and wh-questions. Yes-no questions are formed in the same way as simple sentences, except that they have one auxiliary verb that occurs before, rather than after, the subject noun phrase, for example (20) and (21). Switching the placement of the auxiliary verb and the subject is called subject-aux inversion.

20. Is John enrolled in COMP3160?
21. Do most students work?

The formation of wh-questions involves interrogative words (*wh*-words and *how*). We distinguish in PENG^{ASP} between subject questions and complement/adjunct questions. Subject questions such as (22) and (23) are constructed from a wh-word and a finite verb phrase. Complement/adjunct questions such as (24) and (25) are formed from a wh/how-word that has been moved to the front and acts now as a filler for a gap in a subject-aux-inverted clause.

22. Who is enrolled in COMP3220?
23. Who is not enrolled in COMP3220?
24. What does Liam study?
25. When does the student arrive?

Note that we can answer question (23) in the context of our motivating example in Section 4, after adding the conditional sentence (17) to that specification, since the corresponding ASP rule for (17) ensures that all negative atoms for the enrollment property are derived. Answering the yes-no question in (21) requires a similar mechanism with an agreed threshold for the quantifier *most*.

A special case are questions that ask for an amount like (26) or a quantity like (27). They are formed with the help of a keyphrase and a noun that serve as a filler for a complement gap in a subject-aux inverted clause.

26. [How much] time does Liam spend on the first assignment?
27. [How many] units does Liam attend?

In the case of (26) the keyphrase is followed by an uncountable noun and in the case of (27) by a countable plural noun.

5.5 Issuing Directives

Directives are used in PENG^{ASP} to issue weak constraints in order to prioritise certain solutions. Syntactically, a directive is expressed with the help of a keyphrase like in (28) or (29) that starts with a specific verb in its bar infinitive form, followed by a priority level expressed as a prepositional phrase and a relative pronoun.

28. [Minimise with a priority of 3 that] a student accommodation is noisy.
29. [Maximise with a priority of 2 that] a student accommodation is central.

The description of the statement that is prioritised can have the same syntactic form as the description of a statement in a strong constraint.

6 Implementation Details

The grammar rules for the PENG^{ASP} system are specified in definite clause grammar notation and contain feature structures as arguments. These feature structures have the form of *name:value* pairs; names are Prolog atoms and values are Prolog terms (even compound terms of the form `[H|T]-T` are allowed).

The most important feature names in the grammar are: `mode` for the processing mode; `clause` with an incoming and an outgoing list as values for the assembly of ASP clauses (in the case of processing) and the disassembly of ASP clauses (in the case of generation); `ante` with an incoming and an outgoing list as values for the recording of accessible antecedents; `ctx` with a value (e.g., `fact` indicating a factual statement) for the functional context of a rule; and `fcn` with a value (e.g., `tmod` indicating a temporal modifier) for the structural function of a rule.

The feature structures for the functional context and the structural function of rules allow us to tailor the grammar for application scenarios that do not require the full power of ASP. This means the grammar is highly configurable and we can easily exclude certain linguistic constructions from the grammar if they are not required.

A number of additional syntactic feature names (e.g., `crd`, `num`, `vmode`, and `wform`) and semantic feature names (e.g., `def`, `arg` and `lit`) are used in the grammar; the meaning of these features should become clear in the following discussion.

The implementation of the bi-directional grammar can be best explained with the help of a concrete example. The grammar rules in Listing 4-10 translate a factual statement with a complex temporal modifier like:

30. Rona arrives on 2021-04-24 at 09:15.

incrementally into the following internal ASP representation:

Listing 2: Internal Answer Set Program

```
[['.',
  data_prop(A, 9, 15, time),
  data_prop(A, 2021, 4, 24, date),
  data_prop(A, B, date_time),
  happens(event(C, arrive), B),
  named(C, rona)]]
```

This internal ASP representation is then further translated by the Writer module of the PENG^{ASP} system into an executable ASP program:

Listing 3: Executable Answer Set Program

```
named(1, rona).
happens(event(1, arrive), 1619255700).
data_prop(2, 1619255700, date_time).
```

In the case of verbalising the above-mentioned ASP program, the Reader module reads this executable ASP program and expands it into the internal ASP representation but now in reverse order compared to the representation in Listing 2. The Planner module supports this process, if required, and decides when two or more literals should be aggregated and how these literals should be transformed into the aggregated structure.

6.1 Processing a Specification

For the processing of a specification, the `s`-rule in Listing 4 is used to split the sentence (30) into a noun phrase and a verb phrase. The feature structure `mode:M` takes care of the mode (either `proc` for processing or `gen` for generating). The feature structure `ctx:fact` indicates that this grammar rule is used to deal with a factual statement. The feature structure `clause:C1-C4` consists of a variable `C1` for the incoming list and a variable `C4` for the outgoing list. Remember that this data structure is used to collect the literals for the ASP program during the parsing process. This means the noun phrase takes a list as input and returns a modified list as output as indicated by the feature structure `clause:C1-C2`. This modified list then serves as input to the verb phrase as indicated by the feature structure `clause:C2-C3` and its output as input to the category for the full stop as indicated by the feature structure `clause:C3-C4`, since the full stop is the last element that is added to the front of the outgoing list (as illustrated in Listing 2). In a similar way, the feature structure `ante:A1-A3` collects all accessible antecedents with the help of an incoming and an outgoing list. The feature with the name `tree` takes a list as value and is responsible for constructing a syntax tree; this is in particular helpful for developing the grammar. It is important to note that all the above-mentioned tasks occur in parallel due to the power of unification.

Let us have a closer look at the noun phrase in the `s`-rule: the feature structure `crd:'-'` specifies that this noun phrase cannot be coordinated; the feature structure `fcn:subj` states that the noun phrase occurs in subject position and the feature structure `def:_` indicates that the definiteness of the noun phrase is unspecified in our example.

Listing 4: DCG rule for a factual statement

```
s([mode:M, ctx:fact, clause:C1-C4, ante:A1-A3, tree:[s:17, NP, VP]]) -->
  np([mode:M, ctx:fact, crd:'-', fcn:subj, def:_D, num:N, arg:X, clause:C1-C2,
      ante:A1-A2, tree:NP]),
  vp([mode:M, ctx:fact, crd:'+', num:N, arg:X, clause:C2-C3, ante:A2-A3, tree:VP]),
  fs([mode:M, clause:C3-C4]).
```

Listing 5: DCG rule for a noun phrase in subject position

```
np([mode:M, ctx:fact, crd:'-', fcn:subj, def:'+', num:N, arg:X, clause:C1-C3,
    ante:A1-A3, tree:[np:72, PN]]) -->
  pn([mode:M, wform:_, num:N, arg:X, clause:C1-C2, ante:A1-A2, tree:PN]),
  { anaphora_resolution(pn, [M, '+', X, C1, C2, C3, A1, A2, A3]) }.
```

Listing 6: DCG rules for the lexicon look up of a proper name

```
pn([mode:proc, wform:WForm, num:N, arg:X, clause:[C1|C2]-[[L|C1]|C2],
    ante:[A1|A2]-[[L|A1]|A2], tree:[pn:323, WForm]]) -->
  { lexicon([cat:pn, wform:WForm, num:N, arg:X, lit:L]) }, WForm.

pn([mode:gen, wform:WForm, num:N, arg:X, clause:[[L|C1]|C2]-[C1|C2],
    ante:[A1|A2]-[[L|A1]|A2], tree:[pn:324, WForm]]) -->
  { lexicon([cat:pn, wform:WForm, num:N, arg:X, lit:L]) }, WForm.
```

Listing 7: DCG rule for a verb phrase with a prepositional (temporal) modifier

```
vp([mode:M, ctx:fact, crd:'-', num:N, arg:X, clause:C1-C3,
    ante:A1-A3, tree:[vp:211, VP, PP]]) -->
 vc([mode:M, ctx:fact, num:N, arg:X, hold:[pred(X, PN)]-[happens(event(X, PN), T)],
     clause:C1-C2, ante:A1-A2, tree:VP]),
 pp([mode:M, ctx:fact, crd:'-', fcn:tmod, arg:T, clause:C2-C3, ante:A2-A3, tree:PP]).
```

Listing 8: DCG rule for an intransitive verb

```
vc([mode:M, ctx:fact, num:N, arg:X, hold:L1-L2, clause:C,
    ante:A-A, tree:[vc:217, IV]])  -->
  iv([mode:M, wform:_, num:N, vform:fin, arg:X, hold:L1-L2, clause:C, tree:IV]).
```

Listing 9: DCG rules for the lexicon lookup of an intransitive verb

```
iv([mode:proc, wform:WForm, num:N, vform:V, arg:X, hold:[L1]-[L2],
    clause:[C1|C2]-[[L2|C1]|C2], tree:[iv:325, WForm]]) -->
  { lexicon([cat:iv, wform:WForm, num:N, vform:V, arg:X, lit:L1]) }, WForm.

iv([mode:gen, wform:WForm, num:N, vform:V, arg:X, hold:[L1]-[L2],
    clause:[[L2|C1]|C2]-[C1|C2], tree:[iv:326, WForm]]) -->
  { lexicon([cat:iv, wform:WForm, num:N, vform:V, arg:X, lit:L1]) }, WForm.
```

Listing 10: DCG rule for a prepositional (temporal) modifier

```
pp([mode:M, ctx:fact, crd:'-', fcn:tmod, arg:T, clause:C1-C3,
    ante:A1-A3, tree:[pp:243, Prep1, Date, Prep2, Time]]) -->
 prep([mode:M, wform:[on], tree:Prep1]),
 date([mode:M, ctx:fact, arg:X, arg:T, clause:C1-C2, ante:A1-A2, tree:Date]),
 prep([mode:M, wform:[at], tree:Prep2]),
 time([mode:M, ctx:fact, arg:X, clause:C2-C3, ante:A2-A3, tree:Time]).
```

The feature structure num:N enforces number agreement between the noun phrase and the verb phrase, and the feature structure arg:X ensures that the argument for the noun (or proper name) in the noun phrase becomes available for the predicate-argument structure in the verb phrase. In contrast to the noun phrase that cannot be coordinated, the feature structure crd:'+' in the verb phrase indicates that verb phrase coordination is possible.

The np-rule in Listing 5 first checks whether the noun phrase that occurs in subject position consists of a proper name. For this purpose, the first pn-rule in Listing 6 is used that looks up the word form for the proper name in the lexicon and unifies the variable L for the literal with the corresponding value named(C, rona) that is stored in the lexicon for the proper name. The value for the literal is then added to the outgoing list that is responsible for clause construction (clause); at the same time the outgoing list that records all the accessible antecedents (ante) is updated. Afterwards, the anaphora resolution algorithm in the np-rule in Listing 5 is used to check if the proper name has been previously introduced and is now used anaphorically or not (both the clause lists and antecedent lists are then updated accordingly).

Once this has been done, the vp-rule in Listing 7 takes care of the verb phrase. This rule basically splits a verb phrase into an obligatory part and an optional part. The obligatory part consists in our case of an intransitive verb (without a complement) and the optional part consists of a prepositional phrase that serves as a temporal modifier for the verb. The important point to note here is that the vc-rule transforms a literal for an atemporal specification pred(X, PN) into a literal for a temporal specification happens(event(X, PN), T) with the help of a holding list, once the temporal modifier has been processed. The vc-rule in Listing 8 calls the first iv-rule in Listing 9 that processes the intransitive verb. Note that the variable L1 that represents the literal for the intransitive verb is not immediately added to the outgoing list for the clause in this rule. This variable is first added to a holding list together with a second variable L2 that serves as a placeholder. This second variable is added to the outgoing clause list instead of the first one. This is done because we do not known at this point of processing if the verb will finally be temporally modified or not. This information becomes only available once the pp-rule for the temporal modifier in Listing 10 has been processed. This rule adds three literals to the outgoing clause list (see Listing 2 for details) and completes the processing of the verb phrase.

6.2 Verbalising an ASP Program

Before an ASP program can be verbalised, it needs to be transformed into a linguistically processable version by the Reader module and potentially redundant structures need to be identified and aggregated by the Planner module. The grammar then takes the representation of the ASP program in Listing 2 in reverse order as input. The second pn-rule in Listing 6 removes the literal named(C, rona) from the incoming clause list and adds this literal to the outgoing antecedent list, since it may serve as a potential antecedent later. The same rule then generates the word form *Rona* for the removed literal. The anaphora resolution algorithm of the np-rule in Listing 5, then checks the status of the antecedent. Once the noun phrase has been generated, the second iv-rule in Listing 9 removes the (reduced) literal for the intransitive verb from the incoming clause list with the help of the information on the holding list and generates the verb form *arrives*. In a similar way, the three incoming literals for the temporal modifier are removed from the incoming clause list and generate the word forms that describe the temporal modifier. Generating a full stop terminates this process.

7 Conclusion

PENG^{ASP} is a machine-oriented controlled language designed to specify ASP programs in a natural way. The grammar of PENG^{ASP} is written in definite clause grammar notation and is a bi-directional one. The grammar can be used to translate a specification into an executable ASP program and to generate a semantically equivalent verbalisation of that ASP program. Anaphoric references can be resolved directly during the parsing process, since the anaphora resolution algorithm is tightly integrate with the grammar. The grammar is parameterised using feature structures so that subsets of the grammar can be selected in an easy way for various application scenarios without breaking the grammar. While this paper focuses on the features and coverage of the language and the grammar of PENG^{ASP}; it is important to note that the writing of a specification in PENG^{ASP} is supported by a smart authoring tool.

References

Peter Clark, Phil Harrison, Tom Jenkins, John Thompson, and Rick Wojcik. 2005. Acquiring and using world knowledge using a restricted subset of english. In *Proceedings of FLAIRS'05*, pages 506–511.

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

Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. 2008. Attempto Controlled English for knowledge representation. In *Reasoning Web*, volume 5224 of *LNCS*, pages 104–124. Springer.

Gerald Gazdar and Chris Mellish. 1989. *Natural Language Processing in PROLOG, An Introduction to Computational Linguistics*. Addison Wesley.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele, and Philipp Wanko. 2019. *Potassco User Guide, Version 2.2.0*.

Michael Gelfond and Yulia Kahl. 2014. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents, The Answer-Set Programming Approach*. Cambridge University Press.

Michael Gelfond and Valdimir Lifschitz. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming (ICLP)*, pages 1070–1080.

Stephen Guy and Rolf Schwitter. 2017. The PENG^{ASP} system: Architecture, language and authoring tool. *Journal of Language Resources and Evaluation, Controlled Natural Language*, 51:67–92.

Richard I. Kittredge. 2003. Sublanguages and controlled languages. In Ruslan Mitkov, editor, *The Oxford Handbook of Computational Linguistics.*, chapter 23, pages 430–447. Oxford University Press, Oxford.

Robert Kowalski. 2020. Logical english. In *Proceedings of the 2nd Workshop on Logic and Practice of Programming (LPOP)*, pages 33–37.

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

Erik T. Mueller. 2015. *Commonsense Reasoning: An Event Calculus Based Approach*. Morgan Kaufmann; Second Edition.

Fernando C.N. Pereira and David H.D. Warren. 1980. Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278.

Rolf Schwitter. 2010. Controlled natural language for knowledge representation. In *Proceedings of COLING 2010*, pages 1113–1121. Association for Computational Linguistics.

Rolf Schwitter. 2020. Lossless semantic round-tripping in PENG^{ASP}. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 5291–5293. International Joint Conferences on Artificial Intelligence Organization. Demos.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.