Autoformalisation Answer Set Programs for Scheduling Problems using Few-Shot Learning and Chain-of-Thought: Preliminary Results

Jesse Heyninck[1,2], Bart van Gool[1], Stefano Bromuri[1] and Tjitze Rienstra[3]

[1]Open Universiteit, the Netherlands
[2]University of Cape Town, South-Africa
[3]Maastricht University, the Netherlands

Abstract

Large language models (LLMs) have caused a veritable revolution in the field of AI. However, LLMs do come with some considerable caveats including the lack of logical reasoning ability. This can make it challenging to use LLMs in environments where they need to give reliably correct answers. Recently, attempts have been made to alleviate this concern by generating a more transparent way of solving the problem using an LLM, instead of solving the problem directly with an LLM (so-called autoformalisation). Among others, answer set programs have been tried as a problem-solving intermediary in this context. However, current attempts at autoformalisation of answer set programs has been limited to toy examples or single, simple rules. In this work, we investigate the capabilities of LLMs in generating ASP that solve real-world scheduling problems, and identify techniques such as few-shot learning and chain-of-thought as particularly succesful.

1. Introduction

Data-driven learning techniques such as (large) language models ((L)LMs) have made impressive advances recently. However, their reasoning capabilities are unpredictable and non-transparent, which is especially alarming as LLMs sometimes hallucinate information, leading to unreliable inferences [1, 2, 3, 4].

A fundamentally different approach to AI is *symbolic* AI (SAI), where knowledge is represented in a way so that both humans and computers can reason with it, making reasoning highly reliable and transparent. A paradigmatic example of SAI is *logic programming* (LP) [5], with efficient solvers [6] and a wide showcase of applications [7]. A major bottleneck in SAI's deployment is that of *knowledge acquisition*: a formal representation of the domain knowledge has to be handwritten by an analyst, thus requiring specialised expertise in SAI.

Recently, there have been first attempts in using LLMs to generate formal representations (in the form of LPs) and then using the generated formal representation to perform the reasoning using an LP-solver [8, 9, 10, 11]. This outsources the reasoning responsibilities from an LLM to SAI, making inferences more reliable and transparent, while also alleviating SAI's *knowledge acquisition bottleneck*. While being an important first step towards using LLMs to generate answer set programs, these works have their limitations. Indeed, these approaches perform well on artificial benchmarks consisting of small puzzles, where the relevant background information is given in the description or handwritten in the form of additional LP-rules [10, 11], and often suffer from overfitting on the training data [9], but to the best of our knowledge, no approach performs sufficiently well on real-life applications of ASP. This means that the full potential of this combination of LLMs and SAI has not yet been reached. In this work, we investigate the capablities of LLMs in generating answer set programs for *real world problems*. We do this by restricting attention to a specific class of problems, namely scheduling, for which answer

23rd International Workshop on Nonmonotonic Reasoning
jesse.heyninck@ou.nl (. J. Heyninck); bart.van.gool@icloud.com (. B. v. Gool); stefano.bromuri@ou.nl (. S. Bromuri); t.rienstra@maastrichtuniversity.nl (. T. Rienstra)
https://sites.google.com/view/jesseheyninck (. J. Heyninck)
0000-0002-3825-4052 (. J. Heyninck); 0000-0001-6205-8679 (. S. Bromuri); 0000-0001-9971-3067 (. T. Rienstra)

set programs typically follow the *generate,define and test*-methodology. This allows us to guide the LLM using *chain-of-thought*-techniques. Furthermore, we avoid having to fine-tune the LLM by using *few-shot prompting*. We show that this approach is able to autoformalise ASP programs for real-world problems on the basis of natural language descriptions with reasonable succes, consistently improving on baseline-LLMs. This shows that the approach is a promising avenue for further development.
Outline of the Paper: Related work is surveyed in Section 2, and the necessary background on answer set programming and Large Language Models is discussed in Section 3. Our proposed methodology is discussed in Section 4, and is evaluated in Section 4.2.3, after which the paper is concluded (Section 6). All code and results for this paper can be found on https://github.com/jesseheyninck/SchASPLM.

2. Related Work

In the work of Copollilo et al. (2024) [9], a team looked into how a fine-tuning approach could be used to teach LLMs how to write individual ASP rules. The main challenge for them was the lack of any training data sets which have pairs of natural language descriptions and corresponding ASP representations. To mitigate this, a synthetic dataset was created algorithmically by generating pairs of natural language descriptions and matching ASP rules. Although the results appear promising, our own experimentation with the model show that it is completely overfitted to the training data. When presented with a natural language description of a rule using different vocabulary or sentence structures than those found in the training data, the model is unable to generate matching ASP rules. We give an example:

Example 1. *When prompting the model fine-tuned using the data and methodology of Copollilo et al. (2024) [9] with the following prompt:*

```
Write an ASP program for the following problem. If someone is not guilty, then they
    are innocent.
```

The response of the LLM was the following:

```
Write an ASP program for the following problem. If someone is not guilty, then they
    are innocent. I would prefer that predicate moto with value 10 is not
   associated with "green'. If this occurs, it costs 1 at level 1.
Answer: :~assign(10,"green').[1@1]
```

A quick inspection of the training data reveals that the output is clearly very close to existing data-points.

Even though this might not invalidate the research aims of Copollilo et al. (2024) [9], this clearly invalidates their LLM for writing ASP-programs for real-life problems based on natural language specifications. For this reason, we decided not to investigate this model further in this paper. Furthermore, given the rather limited ammount of ASP-code publicly available (when compared to e.g. python, java or even SAT-based encodings) as well as the non-monotonicity of logic programs (which means that a small change in the code might drastically change the meaning of the code), one might be rather pessimistic about the improvements one can expect from traditional fine-tuning and training techniques. Another downside of this method is the high demand for computational and hence electrical power or expensive cloud computing. In short, it makes sense to explore other techniques for LLM-based translation.

Another attempt to teach ASP to LLMs can be found in the work of Ishay et al. [8] who used in-context learning with few shot prompts to enable LLMs to create ASP encodings for small logic problems. Although the family of problems that could be solved by this system was limited in complexity, the approach showed a lot of promise, as the system was able to solve a wide array of different problems. The use of few-shot prompting is therefore a technique we took up further in this paper. The contribution of this paper w.r.t. the work of Ishay et al. [8] is that we tackle real-life problems, and that we combine few-shot prompting with a chain-of-thought architecture.

Other works [10, 11] use LLMs to extract facts (in ASP-format) from a natural language description, and combine this with a hand-written ASP-program to obtain a hybrid model. This research shows that

LLMs are very efficient at extracting facts from natural language, and the goal of our paper is to see whether the full ASP-program can be generated by LLMs.

3. Background

In this section, we recall the necessary background on Answer set programming (Section 3.1) and Large Language Models (Section 3.3).

3.1. Answer Set Programming (ASP)

In this section we cover the necessary syntax of ASP and the generate,define and test-methodology which will be important in our paper.

3.2. Syntax of Answer Set Programming

Symbolic logic offers a framework for interpretable models by encoding knowledge in structured, human-readable formats. Answer Set Programming combines logic programming with non-monotonic reasoning to represent complex relationships and constraints [5]. ASP programs consist of rules, which in turn are composed of literals (positive or negated atomic formulas) and constraints that define which combinations of literals can be true. A key aspect of ASP is its use of stable model or answer set semantics to determine which sets of ground literals are considered stable solutions to a program. The non-monotonic nature of ASP makes it ideal for representing default rules and exceptions common in real-world domains. ASP is particularly valuable for tasks requiring interpretability. Applications of ASP can be found in many areas, including planning, scheduling, and bio-informatics [7].

In ASP, a program II consists of rules of the form:

$$\mathtt{a_0 :\text{-} a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n}$$

where each $\mathtt{a_i}$ is an atom, and "not" represents negation as failure. In ASP, an atom has the form $p(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where p is a predicate symbol, and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. Terms are either constants, variables, arithmetic terms (i. e., $-\mathtt{t}_1$ or $\mathtt{t}_1 \circ \mathtt{t}_2$ with $\circ \in \{+, -, *, /\}$ wrt. some terms $\mathtt{t}_1, \mathtt{t}_2$), or functional terms (i. e., $\varphi(\mathtt{t}_1, \ldots, \mathtt{t}_m)$ with φ a functor, $\mathtt{t}_1, \ldots, \mathtt{t}_m$ terms, and $m > 0$) [12]. Moreover, we express the arity n of a predicate or function φ as φ/n, and an ASP literal ℓ is either an atom or a default-negated atom. Intuitively,the head of the rule ($\mathtt{a}_0$) is derived if all positive literals in the body ($\mathtt{a}_1$, ..., $\mathtt{a}_m$) are true and all negated literals ($\mathtt{not\ a}_{m+1}, \ldots, \mathtt{not\ a}_n$) are not proven true. An answer set is a minimal set of literals satisfying all program rules without cyclic support. Multiple answer sets represent different possible solutions to the encoded problem. Solvers like clingo [13] can automatically calculate these answer sets. If a rule has an empty body, it is called a *fact*, and if its head is empty, it r is clled a *constraint*.

An atom/rule/program is *ground* if it contains no variables.

We also make use of *choice rule*, i.e. rules of the form

$$\mathtt{l\{a_1, \ldots, a_m\}u :\text{-} a_{m+1}, \ldots, a_n, not\ a_{n+1}, \ldots, not : a_o}$$

with $0 \leq m \leq n \leq o$, $a_1, \ldots, a_o$ being atoms, $0 \leq l \leq u$, and l, u being (optional) lower and upper bounds, respectively. Intuitively, any subset of the head atoms (which complies with the upper and lower bound, if specified) can be included in the answer set (if the body is true).

Furthermore, we use *aggregates* and *optimization statements*. The former are used to reason about minima, maxima, sums, and counts over sets of literals. First, an *aggregate element* g is defined as $g = \mathtt{t}_1, \ldots, \mathtt{t}_m : \ell_1, \ldots, \ell_n$ with $\mathtt{t}_1, \ldots, \mathtt{t}_m$ terms, and $\ell_1, \ldots, \ell_n$ literals. An *aggregate* is then defined as $\#\mathtt{agg}\{g_1, \ldots, g_n\} \prec \mathtt{t}$, with $\#\mathtt{agg} \in \{\#\mathtt{count}, \#\mathtt{min}, \#\mathtt{max}, \#\mathtt{sum}\}$ an aggregate function name, $g_1, \ldots, g_n$ aggregate elements, $\prec \in \{<, >, \leq, \geq, =, \neq\}$ an aggregate relation, and $\mathtt{t}$ a term [12]. Optimization statements allow to select answer sets that are minimal w.r.t. a defined cost function.

We only use a specific form of *minimize statements*, which are of the form `#minimize{`$\mathtt{t}_1, \ldots, \mathtt{t}_m$ `:` $\ell_1, \ldots, \ell_n$`}.`, where $\mathtt{t}_1, \ldots, \mathtt{t}_m$ are terms and $\ell_1, \ldots, \ell_n$ are literals. We refer to a set that complies with the minimization as an *optimal* answer set.

We also use the interval ("..") and pooling (";") operators to abbreviate notation. Intervals let us create multiple instances of a predicate determined by an interval of numerical values, and pooling lets us create multiple instances of a predicate by separating elements by ";".

3.2.1. Generate, Define and Test-Methodology

Many answer set programs are written use the so-called *generate, define and test*-methodology, which means that an answer set program consists of three parts. We illustrate this with a simple example:

```
1 nurse(1..3).     day(1..365).   shift(morning;afternoon; night;vacation).
2
3 {assign(N,X,D) : shift(X)} = 1 :- day(D), nurse(N).
4
5 overworked(N)  :- #count{D : assign(N,vacation,D)}< 30, nurse(N).
6 :- overworked(N), nurse(N).
```

The program above is a toy-version of a program that allows to schedule nurses in a hospital (inspired by Dodaro and Maratea [14]). The first rule (besides the facts on line 1) is a so-called *choice rule* that generates potential schedules by assigning exactly one shift for every day and every nurse. This is the so-called *generate*-part of the program. Some of the potential solutions generated by the choice rule are then weeded out using the *test*-part of the program, making use of definitions from the *define*-part. In this example, the second rule defines which nurses are overworked (since they have less than 30 vacation days), whereas the third rule constitutes the test-part, consisting of a constraint that states no nurse can be overworked according to the schedule.

3.3. Language Models

Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand, generate, and process text and natural language. They are trained on vast amounts of text data using deep learning techniques, particularly using transformer neural networks. LLMs can perform tasks like text generation, translation, summarization, and even code writing by predicting and producing coherent responses based on input. Their ability to recognize patterns in natural language enables them to assist in various applications, from chatbots to content creation and research.

There is a large variety of LLMs available, and they can be deployed in a variety of ways. Furthermore, they can be used "out-of-the-box" or further *fine-tuned*, meaning the weights of the LLM are adapted based on an additional dataset. A downside of this method is the high demand for computational and hence electrical power or expensive cloud computing. Therefore, we opted to use two other techniques, *few-shot prompting* and *chain-of-thought*. In few-shot prompting, [15, 16] a small number of datapoints is provided in a prompt, allowing the model to learn from data without having to manipulate the internal weights in the model. Chain-of-thought (COT) [17, 18] is a technique where the model is prompted to generate intermediate reasoning steps before arriving at a final answer, thus providing structure to the reasoning process

4. Methodology

As discussed in Section 2, related work has demonstrated that it is possible to generate simple ASP programs to solve logic problems [8], however, we wanted to create a more flexible system that could handle real-world problems with an arbitrary amount of rules.

During preliminary experimentation, we quickly understood the utility of limiting our scope to a specific set of structurally similar problems. This allowed us to make assumptions about the structure of the program, which informed the chain-of-thought architecture. Hence, we restrcited attention to

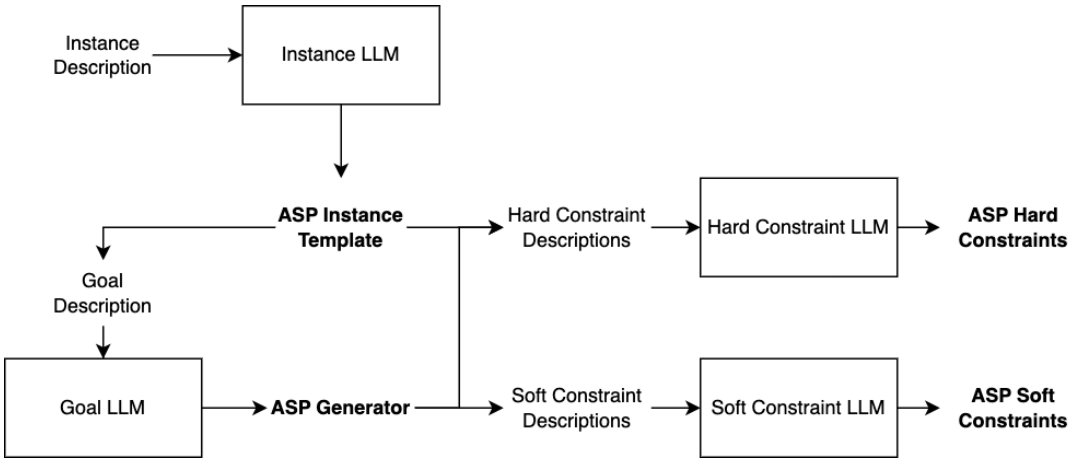

Figure 1: ASP Scheduling Program Generator

scheduling problems, which we believe to be sufficiently specific to make assumptions about the program structure, but still a sufficiently wide and useful class of problems to be of general interest.

We used few-shot learning, to teach the LLMs how ASP rules are constructed. However, to generate a full ASP program, we needed a more complex pipeline which divides the problem into a sequential set of sub-problems and uses intermediate results in downstream generation tasks. As mentioned above, this pipeline was established using the chain-of-thought architecture. The full pipeline, shown in Figure 1, shows in bold where the different parts of the ASP program are generated.

In a nutshell, the pipeline takes as input the problem description in structured natural language. Then, this description is split into relevant parts, and used to prompt an LLM (the "instance LLM") that generates a set of predicates and corresponding terms that will be used to construct the program (the instance template). This instance template, together with the natural language description, is then used to prompt a second LLM (the "goal LLM") that generates one or more choice rules. Similarly, the hard and soft constraints are obtained by prompting two more LLMs. In the following subsections, we describe the parts of this pipeline in more detail.

4.1. Problem Input

The pipeline takes as input the natural language instance description. This input is specifically formatted such that a regular expression powered input parser can split up the input into the necessary sections. Notice that, at the moment, we require quite an extensive description of the problem, including a natural language description of individual constraints and a classification of their type, as well as a description of the relevant variables. We acknowledge this is currently a limitation of our framework, and intend to solve this limitation in future work. The natural language input has five components:

- a *problem description* setting up the general problem. For the examination timetabling problem, for example, this had the following form:

```
Your problem (Examination Timetabling) is a problem that involves making a
    schedule for a set of exams.
Each exam needs to be scheduled during a specific period in a specific room.
It is possible to schedule multiple exams in one room at the same time.
Furthermore, each exam is being taken by a set of students.
```

- a *goal description* describing the generator of the program. For the examination timetabling problem, for example, this had the following form:

```
Goal:
An assignment of exams to periods and rooms.
    - Variables: period, room
```

- a list of *instance variables* describing the predicates that can be used in the program. For the examination timetabling problem, for example, this had the following form:

```
Instance Variables:
- Exams: A set of exams that need to be scheduled.
    - Variables: duration, is_large
- Students: A set of students that are taking a set of exams.
    - Variables: exam
- Periods: A set of periods in which the exams can be scheduled.
    - Variables: date, time, duration, is_late, penalty
- Rooms: A set of rooms in which the exams can be scheduled.
    - Variables: capacity, penalty
- Order_Constrains: A set of constraints that specify the order in which some
   exams must be scheduled.
    - Variables: exam1, exam2
- Same_Time_Constraints: A set of constraints that specify the exams that must
    be scheduled at the same time.
    - Variables: exam1, exam2
- Different_Time_Constraints: A set of constraints that specify the exams that
    must not be scheduled at the same time.
    - Variables: exam1, exam2
- Own_Room_Constraints: A set of constraints that specify the exams that must
   not be scheduled in a room on their own.
    - Variables: exam
```

- a list of *hard constraints* the solutions of the problem have to satisfy, including an indication of their type (regular, sum or count). For the examination timetabling problem, for example, a constraint had the following form (the full list can be found in the appendix):

```
- The number of students taking exams in a room at the same time should not
   exceed the capacity of the room.
    type: count
```

- a list of *soft constraints* the solutions of the problem have to satisfy, in a similar form as the hard constraints. For the examination timetabling problem, for example, a constraint had the following form (the full list can be found in the appendix):

```
- Students should not have more than one exam in the same day
    Penalty: {et_consecutive_penalty} for two exams in the same day that are
   in consecutive periods.
    type: count
```

  Notice that the `{et_not_consecutive_penalty}` is a constant given somewhere else in the file.

4.2. LLM Pipeline

We now describe the LLM pipeline that takes the natural language input and outputs an ASP-program.

4.2.1. Instance Template

Once the input as been split up, the first prompt is created with the goal of generating an instance template. The prompt, found in Listing 1, provides instructions to the LLM as well as an example of how the instance description for curriculum-based timetabling would be turned into an ASP instance template (thus providing a datapoint for few-shot learning). It is crucial that the instance template is generated first because we will use it as part of the prompt in subsequent steps. This is done so that the predicate and variable names used throughout the rest of the program stay consistent.

````
You are a bot that is tasked with turning textual instance data into a set of Answer Set Programming (ASP) facts.
You will be given a set of variables and matching constants, and will turn them into ASP facts.

Here is an example of the input you will receive:

```
Instance Variables:
- Courses: A set of courses that need to be scheduled.
    - Variables: teacher, number of lectures, number of days, number of students
- Rooms: A set of rooms in which the courses can be scheduled.
    - Variables: capacity
- Curricula: A set of curricula, where each curriculum is a set of courses that need to be scheduled in different periods.
    - Variables: courses
- periods: A number of numbered periods in which the courses can be scheduled.
- Days: A set of numbered days in which the courses can be scheduled.
- Unavailabilities: A set of periods on specific days in which a teacher is unavailable.
    - Variables: teachers, days, periods
```

The corresponding ASP facts would be:

```
% Instance Template
course(Course, Teacher, N_lectures, N_days, N_students).

room(Room, Capacity).

curriculum(Curriculum, Course).

period(0 .. N_periods-1).

day(0 .. N_days-1).

unavailability(Teacher, Day, Period).
```

Please provide only the ASP facts in the same format as the example and without any further explanation.
````

Listing 1: Instance Template Prompt

4.2.2. Generator

The next step in the pipeline consists of generating a *choice rule* that servers as a generator. This is done again using a few-shot learning prompt, this time with 3 datapoints. Since generators often have different types of goals, we provide multiple learning examples such that the LLM can get acquainted with the nuances of writing a generator in ASP. Towards the end of the prompt, we also include the instance template generated in the previous step. The full prompt is provided in Listing 2.

```
You are a bot that is tasked with turning textual goal and generator data into a set of Answer Set Programming (ASP) facts.
Given a target goal, you will make an ASP generator that will

```

```
Here is an example of the input you will receive:

"1. An assignment of courses to rooms, days, and periods such that all lectures of
    a course are scheduled to distinct periods.
    - Variables: course, room, day, period
    - Cardinality: N_lectures
2. An assignment of players to teams and positions such that each player is
    assigned to exactly one team and one position.
    - Variables: player, team, position
3. An assignment of tasks to employees and days. Each employee needs needs to be
    assigned to a task on each day."

The corresponding ASP generator would be:

"% 1
N_lectures {{ assigned(Course, Room, Day, Period) : room(Room,_), day(Day), period(
    Period) }} N_lectures :- course(Course,_,N_lectures,_,_).

% 2
1 {{ assigned(Player, Team, Position) : team(Team), position(Position) }} 1 :-
    player(Player).

% 3
1 {{ assigned(Employee, Task, Day) : task(Task) }} 1 :- employee(Employee), day(Day
    )."

Below is a template of an instace for your problem, you may use the predicates and
    variables to construct your generator:
"{instance_template}"

Please provide only the generator in the same format as the example and without any
     further explanation.
```

Listing 2: Generator Prompt

4.2.3. Constraints

Once the instance template and generator have been created, constraints can be generated. Given that programs may contain an arbitrary number of constraints, we have opted to generate them sequentially. This approach helps manage the required context size for each prompt, reducing the likelihood of the LLM overlooking critical details when generating constraints. Much like for the generator prompt, we provide four learning examples in each constraint prompt (as the reader probably gets the idea by now, we have not included examples of these prompts in the paper, but all prompts can be found online).

A key challenge in this process is the presence of multiple constraint types. Broadly, constraints can be categorized as either soft or hard, each requiring distinct syntax. Furthermore, within both categories, additional subtypes exist, including regular constraints and aggregates such as sum and count constraints. To limit the context size and maximize the likelihood that the LLM generates syntactically correct constraints, we designed six distinct constraint prompts. Specifically, for both hard and soft constraints, we created separate prompts, including datapoints for few-shot learning, for regular, sum, and count constraints. The prompts for all these types are given in the appendix. Currently, the type of constraint is given as part of the input. In future work, we plan to investigate how an LLM can identify the type of constraint.

5. Evaluation

We evaluated our pipeline using the following language models: DeepSeek-V3 [19], Meta-Llama-3-8B-Instruct [20] and Meta-Llama-3-70B-Instruct [20]. These models were used in the pipeline described above, but we also queried them in a single prompt to obtain a baseline (i.e. to test the performance of the LLM without the use of few-shot prompting and chain-of-thought). We tested the pipeline using a known scheduling problems from existing literature: Nurse Scheduling [14], and three scheduling problem which have not yet been tackled with ASP in the published literature: Post-Enrollment Based Course Timetabling, sports timetabling and Exam Timetabling[1]. Full problem descriptions as well as the output by the LLM can be found in our online repository.

We evaluated the performance of these models on two aspects: *syntactical correctness* and *semantical correctness*. The former means that the output is within the ASP-language, whereas the latter means that the output correctly formalises the natural language input. The syntactical correctness was evaluated using clingo, and semantical correctness was evaluated manually. We choose this, as opposed to automatically (e.g. by checking the performance of the program w.r.t. some example input facts) as this allowed us to make a more fine-grained assessment of the performance of the models. Using automatic evaluation would potentially allow for one wrong rule or constraint to "spoil" the whole program.

We notice that we have a relatively small amount of evaluation data (four problems), as both the construction of such problems as well as the evaluation is very time-consuming. Setting up automated construction and evaluation methods is an important topic for future work.

The results can be found in Table 1. We listed the percentage (as a fraction, in case the result was not 0 or 1) of rules that were syntactically respectively semantically correct per problem, kind of rule (generator/hard constraints/soft constraints) and model. For example, 2/3 in the column "Ours", "Llama 70b", "Sem", "Sports Timetabling" and the row "Hard Constraints" means that our pipeline, instantiated with Llama 70b, correctly autoformalised 2 out of 3 hard constraints.

We now discuss the results and their implications for our proposed pipeline. Regarding the baseline results, we observe that:

- the small language model *Meta-Llama-3-8B-Instruct* performs terrible at every dataset.
- the large language model *Meta-Llama-3-70B-Instruct* is good at generating syntactically correct logic programs, which are, however, unreliable on the semantic level.
- the large language model *deepseek* is very good at generating syntactically corrrect logic programs, and quite good already at generating logic programs, with the exception of soft constraints.

When looking at how these models perform when used in our pipeline, we see improvements across every model. In more detail:

- the small language model *Meta-Llama-3-8B-Instruct* generates programs that are largely syntactically correct, but are still semantically wrong.
- the large language model *Meta-Llama-3-70B-Instruct* can generate logic programs that are almost entirely syntactically correct, and semantically correct with the exception of a few, quite intricate, constraints.
- the large language model *deepseek* performs comparably to Meta-Llama-3-70B-Instruct.

We notice that these observations hold quite uniformly for the examples on which published ASP-work exists and those on which no ASP-work has been published yet. Thus, we can reasonably rule out that the performance can be explained by pure reproduction of data present in the LLM's training data. In general, we notice that our pipeline leads to a significant improvement w.r.t. the baseline models, and comes close to optimal when instantiated with large language models like Meta-Llama-3-70B-Instruct and Deepseek.

[1]Inspired by the international scheduling competition https://www.eeecs.qub.ac.uk/itc2007/index_files/competitiontracks.htm

	Base line Model						Ours					
	Deepseek		Llama 8B		Llama 70b		Deepseek		Llama 8B		Llama 70b	
	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem	Syn	Sem
	Exam Timetabling											
Instances	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	1	1	1	1	1
Generator	1	0	0	0	1	0	1	1	1	0	1	1
Hard Constraints	1	5/7	0	0	1	6/7	1	5/6	1	0	1	1
Soft Contstraints	1	0/7	0	0	1	1/6	1	4/6	2/6	0	1	1
	Course Timetabling											
Instances	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	1	1	1	1	1
Generator	1	1	0	0	1	1	1	1	1	0	1	1
Hard Constraints	1	1	0	0	1	1	1	6/6	5/6	2/6	1	5/6
Soft Contstraints	1	0/6	0	0	1	2/3	2/3	2/3	2/3	1/3	1	1
	Nurse Scheduling											
Instances	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	1	1	1	1	1
Generator	1	1	0	0	1	0	1	1	1	0	1	1
Hard Constraints	1	6/8	0	0	1/6	0	7/8	7/8	7/8	2/8	7/8	7/8
Soft Contstraints	0	0	0	0	0	0	1/1	1/1	1	0	1	1
	Sports Timetabling											
Instances	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	1	1	1	1	1	1
Generator	1	0	0	1	0	0	1	1	0	1	1	1
Hard Constraints	1	1	0	0	1	1	1	1	1	1	2/3	2/3
Soft Contstraints	1	0	0	0	1	0	1	1	1	1	0	0

Table 1
Evaluation Results for exam timetabling, course timetabling, nurse scheduling and sports timetabling problems.

6. Discussion

In this paper, we presented an approach for the autoformalisation of answer set programs for a variety of scheduling programs, formalised in natural language. The approach makes use of few-shot prompting and chain-of-thought, which is possible due to the assumption that answer set programs make use of the *generate,define and test*-methodology. We believe this is an important step towards solving the *knowledge acquisition bottleneck* in KRR, which can both make KRR easier applicable and LLM-based applications more reliable. While pointing out the promise in our approach, we believe the pipeline can still be approached on several fronts: (1) An obvious avenue is further fine-tuning the prompts and examples used in the few-shot learning prompts. This could be done using sota-techniques for prompt engineering [21], potentially based on reinforcenemt learning [22]. (2) Another limitation we want to alleviate is that we require a rather extensive description of the problem in natural language, which already includes the constraints described in natural language and categorised according to their type. (3) Another opportunity for further work with potentially major impact is situated at the end of the pipeline, by taking inspiration from so-called *feedback-driven code generation* [23] and feeding the output of clingo and debuggers [24, 25] can be fed back to the LLM. Likewise, the semantic correctness of the autoformalised program can be improved by adding a testing component if correct example schedules are present. (4) Even though using ASP-rules and solvers to reason instead of letting the LLM "reason" can be argued to increase explainability and verifiability, this somewhat passes the buck to the autoformalisation of ASP-rules. Alleviating the introduced opacity in this autoformalisation by e.g. combining techniques for explaining LLM-outputs [26] with techniques for explaing ASP is another avenue for future work [27]. Furthermore, of course, the approach can be extended to other problem areas to which ASP can be applied.

Acknowledgments This work was supported by the project LogicLM: Combining Logic Programs with Language Model with file number NGF.1609.241.010 of the research programme NGF AiNed XS Europa 2024-1, is (partly) financed by the Dutch Research Council (NWO).

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to Grammar and spelling check, polish sentences and reword. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content. As the capability of GenAI to generate ASP-programs is the main subject of this paper, GenAI was also used in the experiments, as described in this paper.

References

[1] K. Valmeekam, A. Olmo, S. Sreedharan, S. Kambhampati, Large language models still can't plan (a benchmark for llms on planning and reasoning about change), arXiv preprint arXiv:2206.10498 (2022).

[2] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günnemann, E. Hüllermeier, et al., Chatgpt for good? on opportunities and challenges of large language models for education, Learning and individual differences 103 (2023) 102274.

[3] P. Manakul, A. Liusie, M. J. Gales, Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models, arXiv preprint arXiv:2303.08896 (2023).

[4] L. Berglund, M. Tong, M. Kaufmann, M. Balesni, A. C. Stickland, T. Korbak, O. Evans, The reversal curse: Llms trained on" a is b" fail to learn" b is a", arXiv preprint arXiv:2309.12288 (2023).

[5] V. Lifschitz, Answer set programming, volume 3, Springer Cham, 2019.

[6] M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, P. Obermeier, M. Ostrowski, J. Romero, T. Schaub, S. Schellhorn, P. Wanko, The potsdam answer set solving collection 5.0, KI-Künstliche Intelligenz 32 (2018) 181–182.

[7] E. Erdem, Theory and applications of answer set programming, The University of Texas at Austin, 2002.

[8] A. Ishay, Z. Yang, J. Lee, Leveraging large language models to generate answer set programs, in: Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, 2023, pp. 374–383.

[9] E. Coppolillo, F. Calimeri, G. Manco, S. Perri, F. Ricca, Llasp: fine-tuning large language models for answer set programming, in: Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, 2024, pp. 834–844.

[10] Z. Yang, A. Ishay, J. Lee, Coupling large language models with logic programming for robust and general reasoning from text, arXiv preprint arXiv:2307.07696 (2023).

[11] A. Rajasekharan, Y. Zeng, P. Padalkar, G. Gupta, Reliable natural language understanding with large language models and answer set programming, arXiv preprint arXiv:2302.03780 (2023).

[12] 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. URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.

[13] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot asp solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82.

[14] C. Dodaro, M. Maratea, Nurse scheduling via answer set programming, in: Logic Programming and Nonmonotonic Reasoning: 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings 14, Springer, 2017, pp. 301–307.

[15] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[16] Y. Wang, Q. Yao, J. T. Kwok, L. M. Ni, Generalizing from a few examples: A survey on few-shot learning, ACM computing surveys (csur) 53 (2020) 1–34.

[17] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al., Chain-of-thought

prompting elicits reasoning in large language models, Advances in neural information processing systems 35 (2022) 24824–24837.
[18] Y. Xia, R. Wang, X. Liu, M. Li, T. Yu, X. Chen, J. McAuley, S. Li, Beyond chain-of-thought: A survey of chain-of-x paradigms for llms, arXiv preprint arXiv:2404.15676 (2024).
[19] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al., Deepseek-v3 technical report, arXiv preprint arXiv:2412.19437 (2024).
[20] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, et al., The llama 3 herd of models, 2020.
[21] L. Reynolds, K. McDonell, Prompt programming for large language models: Beyond the few-shot paradigm, in: Extended abstracts of the 2021 CHI conference on human factors in computing systems, 2021, pp. 1–7.
[22] P. Batorski, A. Kosmala, P. Swoboda, Prl: Prompts from reinforcement learning, 2025. URL: https://arxiv.org/abs/2505.14412. arXiv:2505.14412.
[23] Z. Li, Y. He, L. He, J. Wang, T. Shi, B. Lei, Y. Li, Q. Chen, Falcon: Feedback-driven adaptive long/short-term memory reinforced coding optimization system, arXiv preprint arXiv:2410.21349 (2024).
[24] P.-A. Busoniu, J. Oetsch, J. Pührer, P. Skočovskỳ, H. Tompits, Sealion: An eclipse-based ide for answer-set programming with advanced debugging support, Theory and Practice of Logic Programming 13 (2013) 657–673.
[25] M. Gebser, J. Pührer, T. Schaub, H. Tompits, A meta-programming technique for debugging answer-set programs., in: AAAI, volume 8, 2008, pp. 448–453.
[26] P. Sarkar, A. V. Prakash, J. B. Singh, Explaining llm decisions: Counterfactual chain-of-thought approach, in: Advanced Computing and Communications Conference, Springer, 2024, pp. 254–266.
[27] J. Fandinno, C. Schulz, Answering the "why" in answer set programming–a survey of explanation approaches, Theory and Practice of Logic Programming 19 (2019) 114–203.