Explanations for Guess-and-Check ASP Encodings using an LLM (Extended Abstract)

Tobias Geibinger[1][0000−0002−0856−7162], Tobias Kaminski[2][0000−0001−9776−0417], and Johannes Oetsch[3][0000−0002−9902−7662]

[1] Institute for Logic and Computation, TU Wien, Austria
[2] Bosch Center for AI, Renningen, Germany
[3] Department of Computing, Jönköping University, Sweden
tobias.geibinger@tuwien.ac.at, tobias.kaminski@de.bosch.com, johannes.oetsch@ju.se

Answer-Set Programming (ASP) [4] is a rule-based decision making and problem-solving formalism used in many areas; among them are industrial optimisation, knowledge management, or life sciences. Thus, it is of great interest in the context of explainability [2]. To ensure the successful application of ASP as a problem-solving paradigm in the future, it is crucial to investigate explanations for ASP solutions. Such an explanation generally tries to give an answer to the question of why something is, respectively is not, part of the decision produced or solution to the formulated problem.

An approach that addresses this problem is xclingo [1], which wraps around the well-known ASP solver clingo [3] and provides explanations for the atoms in the produced answer set. The method works by having the user annotate the rules of the original ASP program with template sentences conveying the informal meaning of each rule. Those template sentences are then instantiated with the concrete solution and strung together in a derivation-like fashion.

Another related method for computing explanations for clingo programs that addresses the problem of why there is no solution is implemented as part of the clingo-explaid tools (github.com/potassco/clingo-explaid). There, the input is an unsatisfiable clingo program, and the output is a set of constraints that needs to be relaxed to produce a solution.

In this work, we introduce a new approach which also aims at explaining the answer sets of an ASP program. In particular, we focus on so-called *guess-and-check* programs, which intuitively can be split into two parts: rules which generate candidate solutions, and constraints which prune invalid candidates. Such programs are generally found when ASP is used for combinatorial problem solving. In contrast to previous attempts, we use a Large Language Model (LLM) [7] as interface between the user and the explanation component. In particular, the LLM is used to interpret and formalise the questions of the user regarding the provided answer set. For example, a user can ask a contrastive question like "Why is the frame type aluminium and not carbon fiber?" and will receive an answer in natural language. The formalised question is then passed on to the ASP-based explanation component based on techniques described by Herud et al. [5], which in turn provides a formal answer. The latter is then again passed to the LLM, which translates it into natural language.

More precisely, our method relies on a technique for computing *minimally unsatisfiable subsets* (*MUSs*). When the input program is already inconsistent, this technique can be applied directly, and a reason for the inconsistency can be provided to the user in natural language. Otherwise, the user can pose questions based on the answer sets. Based on the type of question, the task of the LLM is to generate assumptions entailed by the question in the form of additional constraints, which usually render the original program inconsistent. An answer to the question can then be extracted from the MUSs of the resulting program.

In general, there can be several possible explanations for a given question, and by default, only one is displayed to the user in natural language.

The approach is implemented in Python and uses clingo via its API as an internal solver.[4] Furthermore, the LLM is queried through the OpenAI API. However, those components are easily exchangeable with another solver and/or LLM.

The main advantages of our method are twofold. First, we heavily utilise natural language, both for receiving the questions regarding the solution from the user and in the answers provided to them. This allows for greater flexibility in what kind of question can be asked and makes the answers more concise and understandable.

The second main advantage is the greater range of types of questions and thus answers that are supported. The tool xclingo [1] only provides explanations of why an atom is in the answer set; clingo-explaid only explains why there is no answer set. Our method handles explanations that are contrastive in the sense that it provides an answer based on why something is included instead of something else. Such explanations answer why a decision has been reached in contrast to a different one, i.e., they answer questions of the form "Why P rather than Q?". It has been argued that contrastive explanations are intuitive for humans to understand and produce, and also that standard "Why" questions contain a hidden *contrast case*, e.g., "Why P?" represents "Why P rather than not P?", and that this is the more appropriate way to answer a "Why" question [6].

Furthermore, integrating the LLM enhances the flexibility of our interfaces for question input and answer output. The LLM enables the classification of different types of input questions, such as conflict explanations (why is there no solution), contrastive explanations (why not A instead of B), and counterfactual explanations (why not B). Additionally, it allows for different levels of control over the answers through three alternative explanation modules for back-translation from ASP to natural language: direct translation with or without prompt engineering, using a template language, or using controlled natural language.

In summary, we developed a new method that can answer user questions about the results of an ASP program in natural language. The approach combines the stability of formal reasoning for deriving explanations with the flexibility of LLMs to make the approach accessible to users without knowledge about ASP. In the future, this integration will also allow for further refinements. For example, the LLM could automatically detect which constraints refer to the program input and which constitute background knowledge, to provide more targeted answers to the user (i.e., when the user is only interested in how the input needs to be changed). This may not be possible on a purely syntactical level.

[4] The code is available under the following link and will be made public in the future: https://owncloud.tuwien.ac.at/index.php/s/brpnfXZWGl8yo38

Acknowledgments. This work was supported by funding from the Bosch Center for AI at Renningen, Germany and has benefited from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440 (LogiCS@TUWien). Tobias Geibinger is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute of Logic and Computation at the TU Wien.

References

1. Cabalar, P., Fandinno, J., Muñiz, B.: A system for explainable answer set programming. In: Technical Communications of the 36th International Conference on Logic Programming (ICLP 2020). EPTCS, vol. 325, pp. 124–136 (2020)
2. Fandinno, J., Schulz, C.: Answering the "why" in answer set programming - A survey of explanation approaches. Theory Pract. Log. Program. **19**(2), 114–203 (2019). https://doi.org/10.1017/S1471068418000534
3. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR **abs/1405.3694** (2014), arxiv.org/abs/1405.3694
4. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proceedings of the Fifth International Conference and Symposium. pp. 1070–1080. MIT Press (1988)
5. Herud, K., Baumeister, J., Sabuncu, O., Schaub, T.: Conflict handling in product configuration using answer set programming. In: Proceedings of the ICLP 2022 Workshops, 2022. CEUR Workshop Proceedings, vol. 3193. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3193/paper2ASPOCP.pdf
6. Miller, T.: Explanation in artificial intelligence: Insights from the social sciences. Artificial Intelligence **267**, 1–38 (2019). https://doi.org/https://doi.org/10.1016/j.artint.2018.07.007
7. Raiaan, M.A.K., Mukta, M.S.H., Fatema, K., Fahad, N.M., Sakib, S., Mim, M.M.J., Ahmad, J., Ali, M.E., Azam, S.: A review on large language models: Architectures, applications, taxonomies, open issues and challenges. IEEE Access **12**, 26839–26874 (2024). https://doi.org/10.1109/ACCESS.2024.3365742, https://doi.org/10.1109/ACCESS.2024.3365742

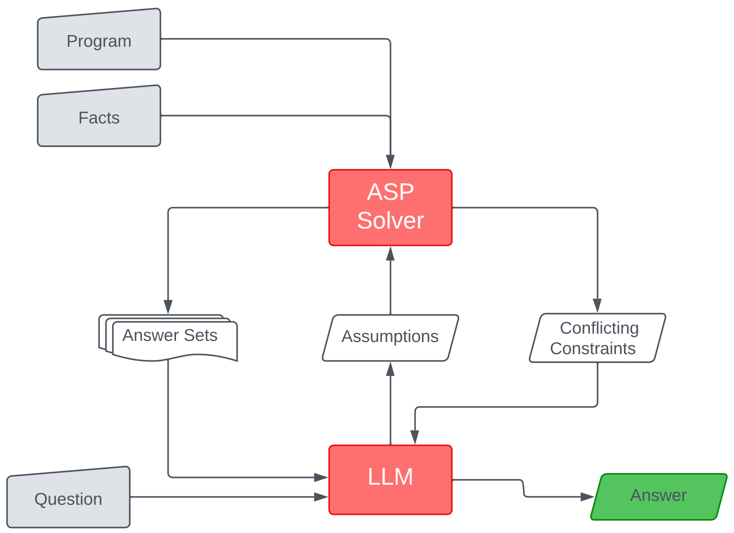

Fig. 1. Overview of our explanation method.

A System Overview

Figure 1 shows an overview of the method, especially how the components interact. It should be noted that the design is modular and the concrete underlying ASP solver and LLM are not fixed.

The individual steps can be described as follows:

1. The basic input is an ASP program that includes input facts. The latter generally represents the input instance data, whereas the program remains largely fixed for a particular problem.
2. The input is then passed to an ASP solver, which is tasked with providing an answer set.
3. If the problem has a solution, the corresponding answer set is printed. Otherwise, we jump directly towards finding a minimally unsatisfiable set of constraints.
4. After that, the user is asked if they have any questions regarding the answer set. Preferably those are questions that are directly contrastive or "why" questions for which the contrast can be inferred.
5. The user question is then passed on to the LLM, which implicitly classifies the type of question, using the information provided in a custom prompt (i.e., conflict explanation, contrastive explanation or counterfactual explanation).
6. Based on the custom prompt, the LLM extracts assumptions about the answer set which corresponds to the question. The custom prompt contains several in-context examples, ASP background, as well as the input, and tasks the LLM to produce ASP constraints, which force certain atoms to be true or false, respectively. Those constraints can be viewed as assumptions on the truth values of these atoms.

Intuitively, if the user asks "Why P and not Q?", the assumptions will enforce a counterfactual where P is false and Q is true.

7. Subsequently, the generated assumptions are added to the original ASP program.
8. With these assumptions in place, the ASP solver is then tasked with finding a minimally unsatisfiable set of constraints for this scenario, thus giving a formal explanation as to why the counterfactual cannot hold.
9. Finally, the LLM is provided with the violated constraints and prompted to translate the constraints into natural language to summarise this information. For this translation, one of at least three alternative approaches can be applied:
   - direct back-translation using the LLM,
   - using a template language to constrain the output, or
   - using controlled natural language to have complete control over the shape of the output.

B Example

Example 1. Consider the following input program, representing the 3-colourability problem with a simple instance of three nodes:

```
% coloring.lp

% choose a color for each node
{ chosenColor(N,C) : color(C) } = 1 :- node(N).

% adjacent nodes are not allowed to have the same color
:- edge(N1,N2), chosenColor(N1,C), chosenColor(N2,C).

% input
color(red).
color(green).
color(blue).
node(1..3).
edge(1,2).
edge(1,3).
edge(2,3).
```

An example execution of the explainer will look as follows:

```
> python main.py examples/coloring.lp
Answer:
chosenColor(3,red) chosenColor(2,green) chosenColor(1,blue)
node(1) node(2) node(3) edge(1,2) edge(1,3) edge(2,3)
color(red) color(green) color(blue)
Question:
why are node 1 and node 2 not both blue?

Explanation:
```

```
Node 1 and node 2 are not both blue because there is an
edge between them, and adjacent nodes are not allowed to
have the same color.
```

In this example, the question “Why are node 1 and node 2 not both blue?” can be reformulated as “Why is node 1 blue and node 2 green in contrast to node 1 blue and node 2 blue?” from which the following assumptions are generated by the LLM:

```
:- chosenColor(1,blue), chosenColor(2,green).
:- not chosenColor(1,blue).
:- not chosenColor(2,blue).
```

The ASP program joined with these assumptions is now inconsistent. The ASP solver is then used to search for a minimal set of constraints from the original program, whose removal restores consistency. In this case, this is achieved by removing an instance of the constraint that expresses that adjacent nodes are not allowed to have the same colour. The LLM finally translates this information into natural language to produce the answer to the initial question.

In order to compute the minimally unsatisfiable set of constraints, we use the following approach. First, the program is rewritten so that new dummy atoms are added to the head of each constraint. In this way, each set of answers will show which constraints have been violated. We can then compute answer sets where the dummy atoms are minimal (either subset minimal or cardinality minimal). Those sets are generally called minimal correction sets. Lastly, we generate a hitting set of all minimal correction sets, which corresponds to a minimally unsatisfiable set of constraints. The approach we employ in this part of our method is largely based on the methods for computing minimal correction sets and minimally unsatisfiable sets described in [5].