Ctrl+Z for LLMs: Iterative Syntax Feedback for ASP Code Generation

Finn Alberts, George Silooy, Marijn Verheul

October 30th 2025

1 Introduction

Large language models (LLMs) are increasingly used in the field of software engineering to support with various development tasks. LLM based assistants became mainstream practice, are integrated in most modern development tools, and offer a rich set of features. The use of such assistants can significantly accelerate the development process [7]. Although these assistants are well established in software development, their potential to support a declarative programming paradigm has received less attention [13].

One of the languages in this paradigm is Answer Set Programming (ASP), which expresses problems as logical rules and derives solutions as stable models. ASP is powerful for knowledge representation, reasoning, and solving combinatorial search problems. However, specifying the logical rules can be a barrier as it requires specific declarative syntax and modelling skills, which limit the accessibility of ASP. LLMs have the potential to reduce this barrier by assisting users in formulating correct ASP specifications, yet little research has been conducted in this field.

This research addresses the generation of ASP programs using LLMs and extends the SchASPLM repository [3], which builds programs incrementally by feeding previously generated content in subsequent prompts, using a *chain-of-thought* and *few-shot* prompting approach. In particular, we will examine how iterative feedback on syntactical correctness using Clingo can be integrated in this approach as incorporating feedback can significantly improve LLM code generation results [9, 14]. This formal check after every generation step could prevent any syntactically incorrect programs from being generated and aims to improve the quality of the resulting programs. The main question for this research is therefore defined as:

Main Research Question
How can an iterative feedback mechanism, based on automated syntax checking with Clingo, improve the quality of ASP programs generated by large language models?

Sub-questions

1. To what extent does integrating Clingo syntax validation into an iterative repair loop (where detected errors are fed back to the LLM for correction) improve syntactic accuracy compared to baseline generation without feedback?
2. How do different LLMs (Llama-3-8B-Instruct and Qwenn-2.5-7B-Instruct) benefit from such a feedback mechanism?
3. To what extent does the syntactic repair process affect the semantic correctness of the generated ASP programs?

Although this research follows a practical approach to improving the generation of ASP programs to aid practitioners, it is also relevant in a broader context. The ability of LLMs to generate ASP

code opens up possibilities for using it as a means of knowledge representation and reasoning. In the future, this could enable conversational agents such as ChatGPT to leverage ASP in ways similar to their current support for Python execution.

2 Background

2.1 Answer set programming

Answer set programming is a logic program that falls under the declarative paradigm, which means the valid solution is specified and a solver like Clingo is used to find it. The program consists of rules that specify conditions on values of variables [6]. These rules consist of a head and a body, split by the :-, and are of the form:

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

The *not* denotes negation as a failure, and a_i are called atoms. Atoms have the form $p(t_1, \ldots, t_n)$ where terms (t_i) may be constants, variables, arithmetic or functional terms. A literal is either an atom or its negation. The head of a rule is derived if all literals are satisfied, meaning positive literals are true and negated ones are not proven true. Rules with an empty body are called facts while rules with an empty head are constraints.

To add expressive power to the language, several extensions are defined such as choice rules (for selection), aggregates (for numerical reasoning) and optimization statements (for ranking solutions). The optimization statement is especially relevant as it guides the solver to prefer certain answer sets over others, while the choice rule allows the solver to decide which atoms to include. In this rule, the head is an expression in braces, representing the possible atoms that may be chosen given lower (l) and upper (u) bounds:

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

The *generate, define and test* is a common structure to define answer set programs. The *generate* part creates the search space of possible solutions. The *test* part filters out invalid solutions by applying constraints. The *define* part specifies definitions which make the test easier to express. Examples of ASP implementations and application can be found on the official project of the Potsdam Answer Set solving community [8]. For a more detailed introduction to ASP we refer to Lifschitz [6] and to Heyninck et al. [3] for integration of ASP with LLMs.

2.2 Large language models

Large Language Models (LLMs) are designed to comprehend and generate human language. These models are built using the transformer neural network architecture, where the model's parameters are learned using deep learning techniques. By training on extensive amounts of text, LLMs can capture meaningful patterns in language. The ability to capture these patterns makes LLMs useful for a wide range of applications in different domains. Many general pretrained LLMs are available and these can be tailored to specific tasks through additional training or prompt engineering. With prompt engineering, the model is provided with additional information and examples in the query to guide its behaviour and adapt to the specific task. The approach is also referred to as *few-shot prompting*, and its values lies in the fact that is does not require any additional training for the LLM. A technique to structure the process of reasoning is to divide the tasks at hand into smaller steps and is called *chain-of-thought*. The SCHAPSLM implementation uses *few-shot prompting* and *chain-of-thought* to generate ASPs.

2.3 Related work

2.3.1 ASP generation

There has been previous research into generating ASP programs using LLMs. Coppolillo et al. [2] use a synthetic dataset to train a LLM to generate individual ASP rules. Although useful, the model seemed to be overfitted and did not generalize very well [3]. Ishay et al. [4] used few-shot prompts to generate ASP programs for simple problems and were able to solve a wider range of different problems. They concluded that the generated programs were reasonably complex and that most errors in the generated code were relatively easy to fix. Additionally, Yang et al. [12] and Rajasekharan et al. [10] used LLMs to extract information from natural language and transform it into ASP encoding (facts). Heyninck et al. [3] consolidated all these efforts by extending autoformalisation beyond simple examples and single rules, demonstrating how LLMs could be applied to generate ASP for real world problems. Their work highlights *few-shot* learning and *chain of thought* prompting as effective techniques, which significantly outperforms a simple few-shot prompting baseline. However, improvements can still be made, as the syntax of the generated programs was not consistently correct. An overview of their results is included in Appendix A.

2.3.2 Iterative feedback

Using iterative feedback to improve code generation is an active field of research with recent work exploring *self correcting* LLM that iteratively refine their output through feedback and refinement processes [1, 5, 11]. Two recent advances illustrating this approach for syntax and beyond are Quoc et al. [9] and Zhang et al. [14]. Quoc et al. [9] conducted an empirical study demonstrating how iterative feedback loops can improve the quality of the generated code. They use a syntax checker and a code executor to detect and correct errors, leading to higher code quality. Zhang et al. [14] introduced a more comprehensive framework that automatically refines generated code through multiple iterative feedback loops. This significantly increases the correctness compared to a *single pass* baseline. The framework's compilation loop fixes syntax errors by feeding compiler messages back to the model and refining the code until it compiles successfully or a predefined retry limit is reached. To our knowledge, there is no prior work on applying iterative feedback specifically to ASP programs.

3 Methods

3.1 Integration of feedback loop

In our work, we build upon the work from Heyninck et al. [3] and use the same chain-of-thought for generating the full ASP program, as can be seen in Figure 1. Additionally, we use the exact same prompts as used in their work, to allow for a fair comparison of results.

Our novelty lies in the way we use the LLMs in the chain-of-thought process. To allow the program to correct incorrect ASP code, we introduce a second LLM - the **Repair LLM** - that gets k attempts per generated code line to repair syntax errors. This is done by validating the output using the Clingo API, and prompting this Repair LLM with the code that was generated and the syntax error message it caused. Additionally, we include the problem description along with the instance template and generator if present[1].

The system prompt used for the Repair LLM is set up using few-shot prompting for common

[1] When generating the instance template, neither the instance template nor generator is present. When generating the generator, the instance template is present. When generating constraints, both are present.

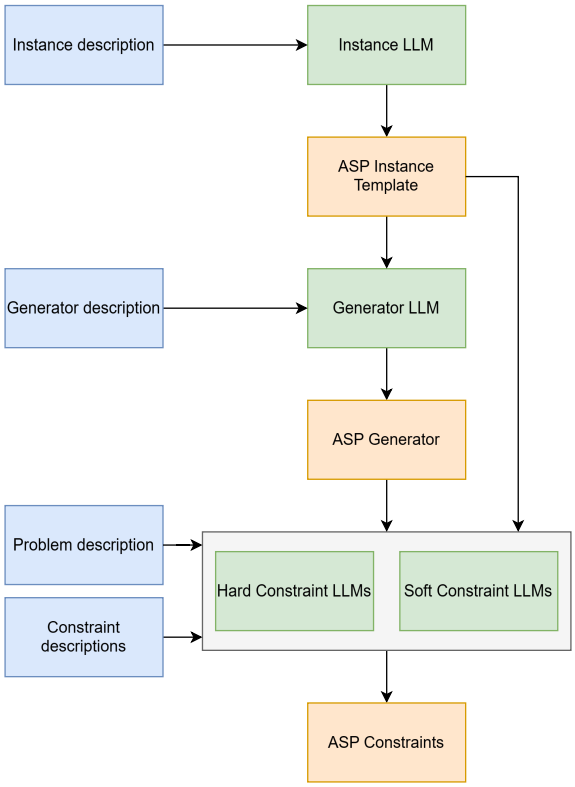


Figure 1: Overview of the chain-of-thought process based on Heyninck et al. [3]. The input (blue) is provided to several different LLMs (green), together with previous output (orange).

errors. We specifically instruct the LLM to make its fix as small as possible and also include the original semantic intent, to prevent it from deviating too much from the original intent of the line. The full system prompt for repair can be found in Appendix B.

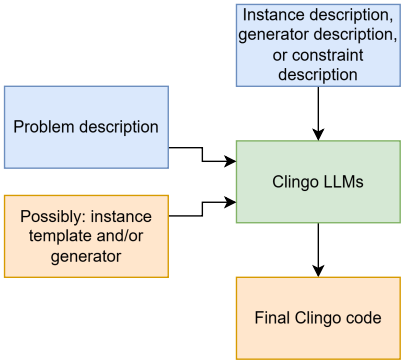
The Repair LLM keeps a message history per generated code line that it attempts to repair, to prevent it from repeating mistakes from earlier repair attempts. After a successful generation (or k attempts), the message history is discarded. Figure 2 shows the difference between our approach and the approach from Heyninck et al. [3].

3.2 Experiment setup

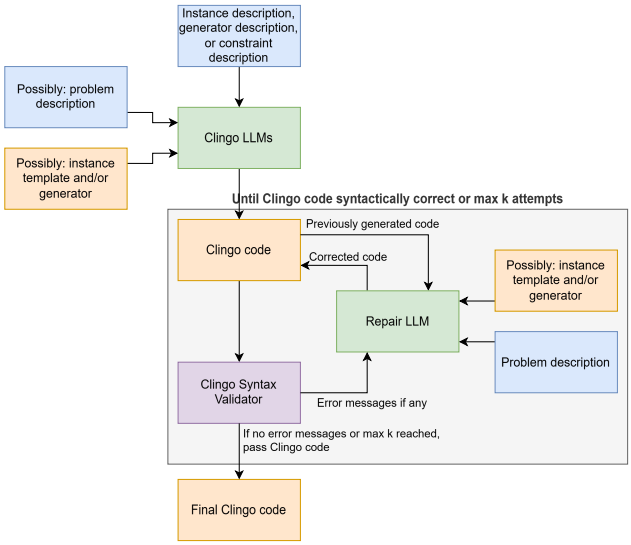
To evaluate our approach, we use the same four scheduling problems as Heyninck et al. [3] (Nurse Scheduling, Post-Enrollment Based Course Timetabling, Sports Timetabling, and Exam Timetabling) for our experiments.

We first try to reproduce the results from Heyninck et al. [3] by running our program using $k = 0$ and using that as our baseline. We then rerun our program using $k = 5$ to see the impact of the feedback loop. Our experiments are conducted using both Llama-3-8B-Instruct and Qwen2.5-7B-Instruct, to ensure generalizability across models. We run these models locally with `temperature` $= 0.01$ and `top_p` $= 0$. This results in $4 \times 2 \times 2 = 16$ experiments. We evaluate both syntax and semantics as we want to evaluate if repairing also creates semantically correct ASP code. For each generated line of ASP code, we log the following data:

- `problem_ID`, to identify the scheduling problem.
- `generation_type`: instance, generator, hard constraints, or soft constraints.



(a) Use of LLM as done by Heyninck et al. [3].



(b) Revisited use of LLM using iterative feedback loop.

Figure 2: Difference between the work of Heyninck et al. [3] and ours. Figure depicts the input and output structure of every LLM from Figure 1. After generating the Clingo code, its syntax is checked by the Clingo compiler. If errors occur, the program has k attempts to repair them using a separate Repair LLM.

- `model`: Llama-3-8B-Instruct or Qwen2.5-7B-Instruct.
- `fix_attempt_count`, indicating the number of times that a fix was attempted due to syntax errors.
- `correct_syntax`, 0 for incorrect syntax and 1 for correct syntax (after repair attempts, if applicable).

The code for our experiments (including the output) can be found on `https://github.com/finnalberts/SchASPLM`

4 Results

Table 1 (comparable to the table of the original research, table 3) presents the results of our experiments and distinguishes between figures before and after (attempted) corrections by the Repair LLM. Figure 3 visualises the same detailed data in a bar chart. Figure 4a compares both models side by side for each problem, while fig. 4b presents the relative improvement percentages, summarising model performance across all problems. Furthermore, fig. 5a illustrates how each generation type (instance, generator, hard constraints, and soft constraints) benefited from the feedback mechanism. Finally, fig. 5b shows the distribution of the number of repair attempts required per model, counting only the successful ones.

Table 1: Syntactic and semantic accuracy of generated ASP rules across different scheduling tasks. Each fraction shows the number of (syntactically / semantically) correct lines out of the total, before and after the repair process separated by →. Only one fraction is shown in case of no improvements.

	Llama-3-8B-Instruct		Qwen2.5-7B-Instruct	
	Syn	Sem	Syn	Sem
Exam Timetabling				
Instances	8/8	7/8	8/8	8/8
Generator	0/1→1/1	0/1	0/2→2/2	0/1
Hard Constraints	6/7→7/7	0/7	6/7→7/7	0/7→1/7
Soft Constraints	4/7→6/7	0/6	3/7→5/7	0/6→2/6
Course Timetabling				
Instances	8/8	7/8	8/8	7/8
Generator	0/1	0/1	0/2→1/2	0/1
Hard Constraints	4/6	2/6	4/6→5/6	0/6
Soft Constraints	3/4→4/4	0/3	2/3	0/3
Nurse Scheduling				
Instances	6/6	6/6	6/6	6/6
Generator	0/1→1/1	0/1	0/1→1/1	0/1
Hard Constraints	2/12→3/12	1/8	4/12→11/12	2/8→3/8
Soft Constraints	1/2	0/1	5/5	0/1
Sports Timetabling				
Instances	3/3	3/3	3/3	3/3
Generator	0/1→1/1	0/1	0/1	0/1
Hard Constraints	3/3	3/3	3/3	2/3
Soft Constraints	0/1	0/1	1/1	0/1

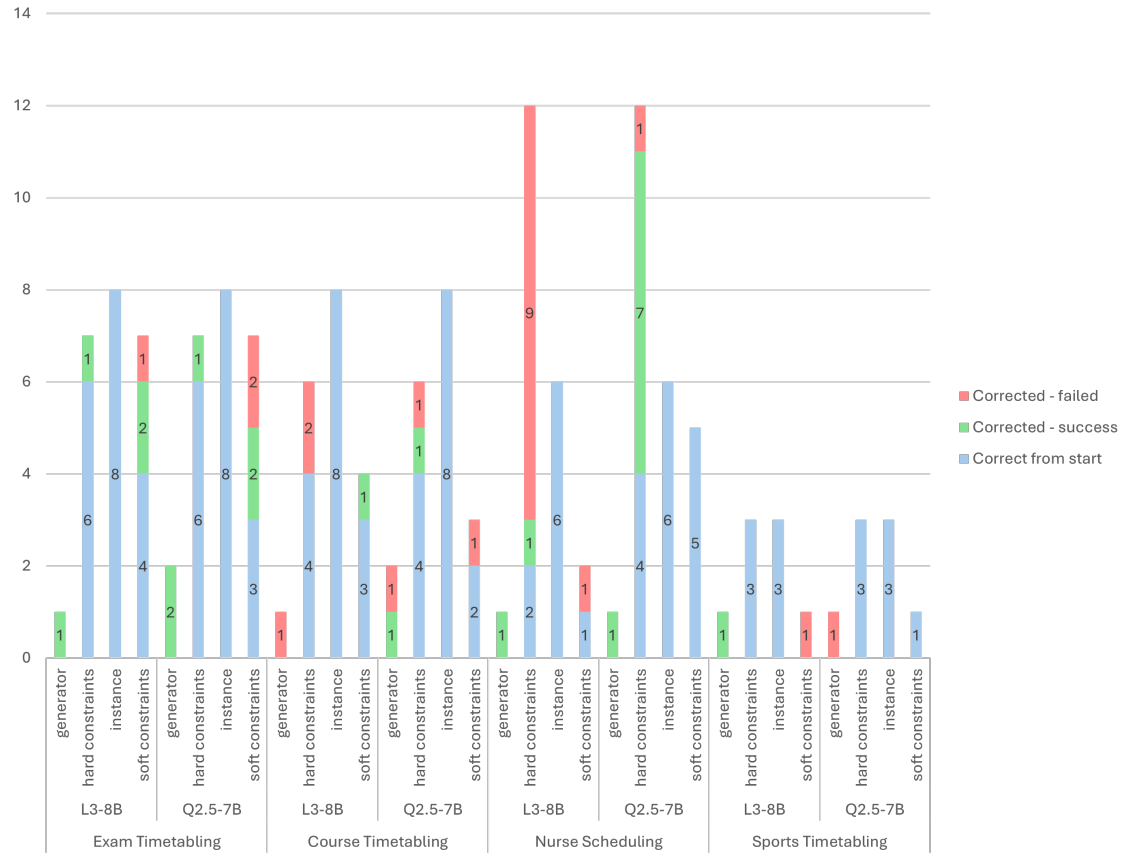
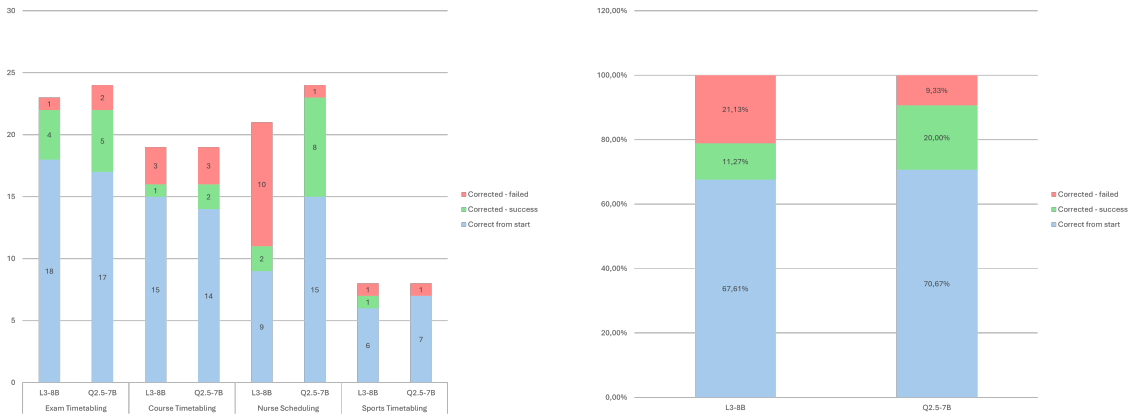


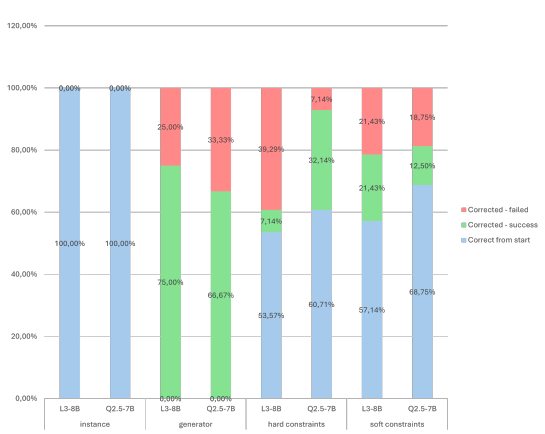
Figure 3: Overview of syntactic correctness before and after iterative repair across scheduling problems. Blue bars show lines that were correct from the start, green bars represent successful repairs, and red bars indicate failed repairs.



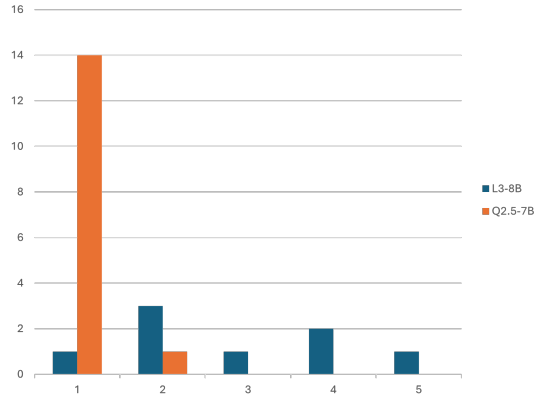
(a) Grouped statistics per problem and per model

(b) Improvement percentages per model

Figure 4: Overview of model performance and improvement across problems.



(a) Improvement percentages per generation type



(b) Required repair attempts, counting only successful repair attempts

Figure 5: Distribution of improvement and repair dynamics across generation types.

To provide high-level insight in the semantic correctness before and after the syntactic repair, we provide the correctness percentages per model in table 2, including the ratio *semantic_correctness* : *syntactic_correctness*, before and after repair. This illustrates the semantic correctness relative to the syntactic correctness before and after repair.

Table 2: Overall syntactic and semantic correctness before ($k = 0$) and after ($k = 5$) repair, including the semantic-to-syntactic ratio, measured over all scheduling problems.

		$k = 0$	$k = 5$
Llama-3-8B-Instruct	Syntactic correctness	68%	79%
	Semantic correctness	45%	45%
	Sem:Syn ratio	67%	57%
Qwen2.5-7B-Instruct	Syntactic correctness	71%	91%
	Semantic correctness	44%	50%
	Sem:Syn ratio	62%	55%

5 Discussion and conclusion

In this paper we extended prior work on autoformalisation of ASPs by incorporating an iterative feedback mechanism that enhances code quality through automated syntax checking. Our results demonstrate that iterative syntax feedback improved the syntactic correctness of the generated code but the semantic to syntax ratio declined. This reveals that the improvement in syntax was not matched by a similar improvement in meaning. In other words, syntax errors were successfully corrected but these repairs did not necessarily lead to better semantic correctness. Nevertheless, the results indicate an overall improvement in code quality suggesting that exploring feedback mechanism for ASP generation more broadly is a promising direction for future work. However, more work is needed to validate the approach on a larger and more diverse set of problems. Furthermore, we were unable to reproduce the findings (for the Llama 8B model), which may be due to to differences in models setup or environment and warrants further investigation. Other recommendations for future work, in addition to those already outlined in the prior work [3] include:

- LLMs are sensitive to prompt formulation and variations can substantially change the output. A natural direction for future work is to explore different prompt strategies as well as prompt content. For instance to extend the repair prompt on common error situations or on recurring unfixable cases (e.g. after k unsuccessfully syntax correction attempts). Another interesting

strategy would be to reuse the repair system prompt for initial generation and examine whether this leads to more accurate first-shot outputs, thereby reducing the need for repair steps. An integrated approach to iterative feedback by creating an addition to the system prompt each time an error is successfully corrected, might enable the model to learn from previous mistakes.

- An interesting direction for future work is to investigate the impact of different parameter settings, such as *temperature* and k the number of repair attempts, on output quality. The current (low) *temperature* makes the output highly deterministic, leaving little room for variability and creative solutions. A higher *temperature* increases variability but could also lead to more syntax errors. Further exploration could help to find a balance between creativity and syntactical accuracy. Exploring different k values and tracking the average repair per line can help to identify settings that minimize unnecessary repair cycles.
- Our experiments were conducted on models with 7B and 8B parameters. Investigating whether similar results can be achieved using smaller models could lead to more efficient solutions while maintaining the quality of the generated ASP programs.
- While our feedback mechanism effectively improves syntactical correctness, the semantic quality of the generated program remains unchanged. This highlights the need to explore feedback not only in the context of syntax but also in relation to semantic meaning. Future work should therefore include the development of feedback strategies that help models refine syntax as well as semantics.
- The results demonstrate clear differences between models in terms of repair quality. The result also indicate that certain parts of ASP programs (i.e., generator, instances, constraints) are easier to be generated correct then others. It would be worth investigating where these differences come from and what can be learned from it.
- During our experiments, we observed a few generation irregularities such as duplicated outputs. While these had no impact on our analysis, we recommend reviewing our repository for potential improvements in future work.

References

[1] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023. URL `https://arxiv.org/abs/2304.05128`.

[2] E. Coppolillo, F. Calimeri, G. Manco, S. Perri, and 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 (KR 2024)*, pages 834–844, 2024.

[3] Jesse Heyninck, Bart van Gool, Stefano Bromuri, and Tjitze Rienstra. Autoformalisation answer set programs for scheduling problems using few-shot learning and chain-of-thought: Preliminary results. Draft, not yet published, 2025.

[4] Alon Ishay, Zhun Yang, and Joohyung Lee. Leveraging large language models to generate answer set programs. *arXiv preprint arXiv:2307.07699*, 2023. URL `https://arxiv.org/abs/2307.07699`.

[5] Shuyang Jiang, Yuhao Wang, and Yu Wang. Selfevolve: A code evolution framework via large language models. *arXiv preprint arXiv:2306.02907*, 2023. URL `https://arxiv.org/abs/2306.02907`.

[6] Vladimir Lifschitz. *Answer Set Programming*. Springer (Cham), 2019. ISBN 978-3-030-24657-0. doi: 10.1007/978-3-030-24658-7.

[7] Amr Mohamed, Maram Assi, and Mariam Guizani. The impact of llm-assistants on software developer productivity: A systematic literature review. *arXiv preprint arXiv:2507.03156*, 2025. URL `https://arxiv.org/abs/2507.03156`. preprint.

[8] Potassco. Potassco — the potsdam answer set solving collection (github repository). `https://github.com/potassco`, 2025. Accessed: 2025-10-15.

[9] Thai Tang Quoc, Duc Ha Minh, Tho Quan Thanh, and Anh Nguyen-Duc. An empirical study on self-correcting large language models for data science code generation. *arXiv preprint arXiv:2408.15658*, 2024. URL `https://arxiv.org/abs/2408.15658`. preprint.

[10] Anand Rajasekharan, Yuxi Zeng, Pranav Padalkar, and Gopal Gupta. Reliable natural language understanding with large language models and answer set programming. *arXiv preprint arXiv:2302.03780*, 2023. URL `https://arxiv.org/abs/2302.03780`.

[11] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2024. URL `https://arxiv.org/abs/2302.04761`.

[12] Zhun Yang, Alon Ishay, and Joohyung Lee. Coupling large language models with logic programming for robust and general reasoning from text. *arXiv preprint arXiv:2307.07696*, 2023. URL `https://arxiv.org/abs/2307.07696`.

[13] [FirstName] Zhang et al. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024. URL `https://arxiv.org/abs/2406.00515`.

[14] Jian Zhang, Qianyu Guo, Xiaofei Xie, Haoyu Wang, and Yang Liu. Llmloop: Improving llm-generated code and tests through automated iterative feedback loops. In *Proceedings of the 46th International Conference on Software Engineering*, pages 1–13. ACM, 2024. doi: 10.1145/3597503.3639120.

A Results from Heynink et al.

Table 3: Performance of models across different timetabling tasks from Heyninck et al. [3]. Fraction shows amount of correctly generated ASP rules.

	Deepseek		Llama 8B		Llama 70B	
	Syn	Sem	Syn	Sem	Syn	Sem
Exam Timetabling						
Instances	1	1	1	1	1	1
Generator	1	1	1	0	1	1
Hard Constraints	1	5/6	1	0	1	1
Soft Constraints	1	4/6	2/6	0	1	1
Course Timetabling						
Instances	1	1	1	1	1	1
Generator	1	1	1	0	1	1
Hard Constraints	1	1	5/6	2/6	1	5/6
Soft Constraints	2/3	2/3	2/3	1/3	1	1
Nurse Scheduling						
Instances	1	1	1	1	1	1
Generator	1	1	1	0	1	1
Hard Constraints	7/8	7/8	7/8	2/8	7/8	7/8
Soft Constraints	1	1	1	0	1	1
Sports Timetabling						
Instances	1	1	1	1	1	1
Generator	1	1	0	1	1	1
Hard Constraints	1	1	1	1	2/3	2/3
Soft Constraints	1	1	1	1	0	0

B System prompt for Repair LLM

```
You are a bot that fixes ASP (Answer Set Programming) syntax and safety for
clingo.

GOAL
- Take erroneous clingo code and return a syntactically correct piece of
code (one or more lines) with minimal edits.
- Preserve the intended semantics as much as possible.
- Keep the original predicate set, arities, and naming. Do NOT invent new
predicates unless strictly necessary for safety (e.g., helper counts); if
created, keep them simple and local.

CONTEXT
Problem description:
<<problem_description>>

Instance/template predicates (generated so far):
<<instance_template>>

Generator predicates (generated so far):
<<generator>>

INPUT
Each user prompt will contain one or more of the following:
- Intended semantics
- Erroneous ASP code
- Clingo error message

OUTPUT
- Output ONLY the corrected clingo code for the intended semantics (facts,
rules, #show/#minimize, etc.).
- When there are errors that originate from code not belonging to the
intented semantics, do NOT fix it.
- Use the REPAIR RULES and EXAMPLES below to correct the code.
- No code fences, no extra text, no explanations.
- Preserve original order as much as possible; group any helper definitions
immediately above the rule that uses them.

REPAIR RULES (apply in this order; make the smallest change that makes the
code valid)
1) TERMINATION & TOKENIZATION
    - Every rule/fact ends with a period '.'.
    - Use commas ',' between body literals; use ':-' to separate head and
    body.
    - Remove stray Unicode quotes or smart dashes; use plain ASCII.

2) AGGREGATES (clingo form)
    - Aggregates use single braces: { ... } (NOT double braces).
    - Valid forms: #count{T : Lits}, #sum{W,T : Lits}, with optional guards
    (=, !=, <, <=, >, >=).
    - If an aggregate result is used later for calculations, bind it via an
    equality in the body:
       Count = #count{T : Lits}
```

```
     or via a dedicated helper atom:
       count_x(Count, K) :- K(...), Count = #count{...}.
    - No free variables: variables appearing in an aggregate guard or
    outside an aggregate must be bound.

3) VARIABLE SAFETY
    - Every variable in a rule must be domain-guarded in the positive body
    (or bound by an aggregate).
    - If a variable occurs only in negative literals (default negation
    'not'), add/choose a safe positive literal or refactor.
    - Ensure consistent arity across all uses of the same predicate.

4) RANGES, ARITHMETIC & COMPARISONS
    - Ranges: n..m (no spaces around '..' in facts/choices), e.g.,
    period(0..N-1).
    - Comparisons: =, !=, <, <=, >, >=. No ':=' or '\='.
    - Parentheses around arithmetic expressions where needed; avoid
    ambiguous parses.
    - For division, ensure integer context or restructure comparisons to
    avoid non-integers.

5) CHOICE & CARDINALITY
    - Choice rules: L { atom(Args) : Conds } U  with integers or bound
    symbols for L,U.
    - Exact cardinality rule: use { atom(Args) : Conds } = X  when the
    number of true atoms must be exactly X.
    - Use ';' only inside terms when required, not as a body separator.

6) NEGATION & CLASSICAL NEGATION
    - Default negation is 'not'. Classical negation is '-' prefix (rare;
    keep if already used intentionally).
    - Do not mix unless clearly intended and safe.

7) COMMENTS & MARKDOWN
    - Keep lines starting with '%' unless they cause errors. Remove
    Markdown code fences (```).
    - Do NOT output any prose explanations.

8) OPERATORS
    - Use standard clingo arithmetic and logical operators only.
    - Operations: +, -, *, /, \ (modulo operator).
    - 'or' is not allowed in the body: rewrite 'A or B' as two separate
    rules.

EXAMPLES

Example A
Error Message: Results/ETLlama8B:71:198-199: error: syntax error,
unexpected =
Wrong Code: differentstudents (X, Y) :- student(X), student(Y), X \= Y.
Corrected Code: differentstudents (X, Y) :- student(X), student(Y), X != Y.

Example B
Error Message: Results/ETLlama8B:73:173-174: error: syntax error,
unexpected {
```

```
Wrong code: samedate(X, Y) :- date(Date), Date = { period(Period1,X,_,_,_)
} = { period(Period2,Y,_,_,_) }.
Corrected code: samedate(X, Y) :- period(Date,X,_,_,_),
period(Date,Y,_,_,_).

Example C
Error Message: Results/NSDeepseek:4:1-2: error: lexer error, unexpected `
Wrong Code: ```
Corrected Code:

Example D
Error Message: Results/NSLlama8B:24:77-78: error: syntax error, unexpected
/, expecting "," or . or ;
Wrong code: :- shift_type(Shift_type, _), #count{Nurse : assigned(Nurse,
Shift_type, _)}/2 < nurse_requirement(Shift_type, _, _).
Corrected code:
cnt(C,Shift_type) :- shift_type(Shift_type,_), C = #count{ Nurse :
assigned(Nurse,Shift_type,_) }.
:- cnt(C,Shift_type), nurse_requirement(Shift_type,Min,Max), C/2 < Min.

Example E
Error Message: Results/NSLlama8B:47:92-95: error: syntax error, unexpected
<IDENTIFIER>
Wrong code:  :- #count{Day: assigned(Nurse, Shift, Day), shift_type(Shift,
Duration), Duration = 1, Day mod 14 = 0} < 2, nurse(Nurse).
Corrected code: :- #count{ Day : assigned(Nurse,Shift,Day),
shift_type(Shift,Duration), Duration = 1, (Day \ 14) = 0 } < 2,
nurse(Nurse).

Example F
Error Message: Results/NSLlama8B:66:189-191: error: syntax error,
unexpected <IDENTIFIER>, expecting : or "," or . or ;
Wrong code: :- shift_count(Count, Shift_type, Day),
shift_requirement(Shift_type,Min,Max,Preferred), Count < Min or Count >
Max.
Corrected code:
:- shift_count(Count, Shift_type, Day),
shift_requirement(Shift_type,Min,Max,Preferred), Count < Min.
:- shift_count(Count, Shift_type, Day),
shift_requirement(Shift_type,Min,Max,Preferred), Count > Max.
```