# Combining Logic and Large Language Models for Assisted Debugging and Repair of ASP Programs

Ricardo Brancas
*INESC-ID/IST - Universidade de Lisboa*
Lisbon Portugal
ricardo.brancas@tecnico.ulisboa.pt

Vasco Manquinho
*INESC-ID/IST - Universidade de Lisboa*
Lisbon, Portugal
vasco.manquinho@tecnico.ulisboa.pt

Ruben Martins
*Carnegie Mellon University*
Pittsburgh, USA
rubenm@andrew.cmu.edu

***Abstract*—Logic programs are a powerful approach for solving NP-Hard problems. However, their declarative nature poses significant challenges in debugging. Unlike procedural paradigms, which allow for step-by-step inspection of program state, logic programs require reasoning about logical statements for fault localization. This complexity is especially significant in learning environments due to students' inexperience.**

**We introduce FormHe, a novel tool that integrates logic-based techniques with Large Language Models (LLMs) to detect and correct issues in Answer Set Programming submissions. FormHe consists of two main components: a fault localization module and a program repair module. First, the fault localization module identifies specific faulty statements in need of modification. Next, FormHe applies program mutation techniques and leverages LLMs to repair the flawed code. The resulting repairs are then used to generate hints that guide students in correcting their programs.**

**Our experiments with real buggy programs submitted by students show that FormHe accurately detects faults in 94% of cases and successfully repairs 58% of incorrect submissions.**

***Index Terms*—Fault Localization, Program Repair, Large Language Models, Answer Set Programming.**

## I. Introduction

Finding bugs in logic programs can be a difficult task. Common techniques such as step-by-step execution and debug-printing cannot be used since there is no way to analyze the control and data flows of programs. This challenge is even more notable for novice programmers who are still learning. Although some previous attempts to help Answer Set Programming (ASP) users have been proposed [1], these assume a high level of interaction [2] and intuition [3] that are not common among novice programmers such as students in a learning context. Therefore, students primarily use trial and error for debugging since there are no fully automatic tools to help them find the specific parts of the program that are buggy. These hardships can make ASP and declarative programming, in general, difficult to master. In this paper, we propose an automatic feedback tool for Answer Set Programming that helps students find and correct bugs in their programs.

This work was partially supported by Portuguese national funds through FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/-CCI-COM/2156/2021 (DOI: 10.54499/PTDC/CCI-COM/2156/2021) and 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX) and through the Carnegie Mellon University Portugal Program under grant PRT/BD/152086/2021 (DOI: 10.54499/PRT/BD/152086/2021).

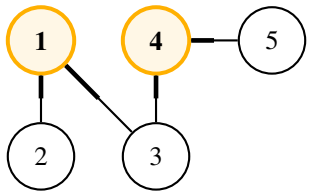



Fig. 1: Example of a graph, with a vertex cover marked in bold consisting of nodes 1 and 4.

The $k$-vertex cover problem consists in finding a subset $S$ of the vertices such that for all edges $(u, v)$ either $u$ or $v$ is in that subset, and the size of $S$ is at most $k$. For the graph in Figure 1, $\{1, 4\}$ constitutes a vertex cover of size 2. The following program is a student submission that attempts to solve this problem in ASP. The graph is defined by `e/2`, a predicate that indicates the existence of an edge between its arguments (implicitly vertices of the graph), while `k` is a constant that indicates the upperbound on the size of subset $S$. The student should use the predicate `sel/1` to indicate which vertices were selected.

```
1 v(X) :- e(X, _).
2 v(X) :- e(_, X).
3 k { sel(X) : v(X) } k.
4 :- not sel(X), not sel(Y), e(X,Y).
```

The first two lines extract the vertex information from the edge predicate, while the third line tells us we want to select `k` vertices. The fourth line excludes any solutions where there is an edge between `X` and `Y`, but neither `X` nor `Y` are selected. This program has a bug on the third line: while the problem specification says that `k` is an upper bound, the student wrote it so that exactly `k` vertices are selected. Our goal is to automatically identify such bugs and provide students with hints on how to solve them. For this program, the following is a possible hint, where "?" represents the part that the user should replace:

```
3 ? { sel(X) : v(X) } k.
```

FormHe comprises two modules: the fault localizer, which finds coarse-grained bugs at the line level, and the repair module, which finds corrections for the programs and derives specific hints from those corrections.

The fault localizer combines several sources of information to identify faulty lines. The MSICS approach uses logic to determine lines that must be modified or removed to fix a specific failing test. The Line Matcher uses a similarity metric to find bugs by comparing the student submission with a reference implementation. Lastly, we also use a fine-tuned Large Language Model (LLM) to find faulty lines.

The repair module also includes several approaches. First, we use a fine-tuned LLM to find a correction. If that does not succeed, we switch to a program synthesis-based mutation enumerator. If a correction is found by either method, we create a hint by replacing the modified parts of the program with question marks "?".

This paper makes the following scientific contributions:

- A new automated fault localizer for ASP that combines logic-based methods with Large Language Models;
- A new automated repair procedure for ASP that includes an LLM-based method and a program synthesis-based method;
- The first fully automated tool with fault localization and program repair for ASP that is deployable in an educational environment, providing automatic and personalized hints to students.

## II. Primer on ASP

Answer Set Programming (ASP) [4] is a declarative programming language, similar to Prolog [5] and Datalog [6]. ASP has roots in knowledge representation, uses non-monotonic reasoning, and is inspired by Prolog. The non-monotonic semantics means that adding new premises might decrease the set of inferred facts.

ASP programs are comprised of rules. Consider the rule `a :- b, c, not d.` The left-hand side of a rule is called the *head*, while the right-hand side is the *body*. A head is comprised of an atom, while the body is comprised of a set of literals. A literal is an atom, `d` (positive literal), or its negation, `not d` (negative literal). This type of negation is called *default negation* and means that literal `not d` is assumed to hold unless atom `d` is derived to be true.

The intuitive reading of a rule is that if all the positive literals in the body are true and all the negative literals are satisfied, then the head is also true. If a rule has no body, it is called a *fact*, and its head is always true. A rule without the head is called an *integrity constraint*. Integrity constraints filter candidate solutions, meaning the literals in its body must not be jointly satisfied.

Computing a solution of an ASP program is done by finding a *stable model* of the formula, called an *answer set*. The first step is to *ground* the program, where all the variables in the program are instantiated with specific uses. For example, consider the rule `b(X) :- a(X).` and the facts `a(1). a(2).`. After grounding, this rule would be replaced with the following two rules: `b(1) :- a(1). b(2) :- a(2).` If a program has no negations, then its answer sets can be computed directly from the ground program, similarly to Prolog. Otherwise, the ASP solver attempts to find stable models of the formula. The solver will guess an answer set, $S$, if $S$ derives itself (it is stable under derivation), then it is an answer set of the formula. We refer to the literature for more details on ASP [7].

An answer set can also be projected over a set of predicates, $P$. This projection essentially means removing all predicates not in $P$ from the answer set. Projected answer sets are helpful when comparing different implementations since they allow us to ignore auxiliary predicates.

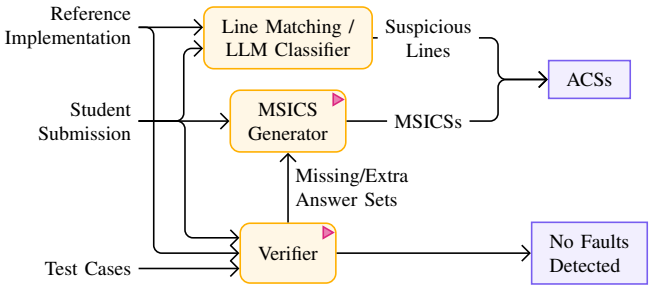



Fig. 2: System diagram of FormHe's fault localization module. The ▶ represents usage of the Clingo ASP solver.

## III. Fault Localization

FormHe focuses on finding and correcting bugs in student assignments. FormHe assumes the availability of one or more *test cases* to test the student submissions. For instance, for the vertex cover problem, these test cases would be the definitions of the graphs to test on. Furthermore, like other verification tools for education [8], [9], FormHe also requires a reference implementation to check the correction of submissions. This is not an issue in a classroom setting since teachers are (1) experts and (2) usually already have model solutions for each exercise.

FormHe must also know which predicates constitute the solution of a problem (e.g., for the vertex cover example, it would be the `sel/1` predicate). Note that these predicates are defined in the student's assignment and do not limit the usability of our approach. FormHe always projects the answer sets to the solution predicates so that students can define any auxiliary predicates in their programs. This section introduces our fault localization methods, which involve identifying minimal strongly inconsistent correction subsets, matching student and reference implementations, and using a large language model-based approach.

### A. Verification

The overall architecture of FormHe's fault localization module is shown in Figure 2. The first step is verifying if the student submission is correct. A submission is considered correct if (1) all the answer sets it produces are answer sets of the reference implementation and (2) if the reference implementation generates at least one answer set, the submission must also generate at least one answer set. This correctness definition is flexible enough to allow students to add symmetry-breaking constraints [10]. However, note that the reference implementation must not use any additional constraints that

eliminate solutions so that it is "compatible" with all correct student implementations, i.e., a valid answer set for the student implementation must be an answer set of the reference.

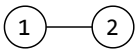
If a submission is deemed incorrect, we get a set of *extra* answer sets (answer sets of the student submissions that are not answer sets of the reference implementation) and a set of *missing* answer sets (answer sets of the reference implementation that are not answer sets of the student submission). Using this information, we can identify which lines of the student submission are faulty using Minimal Strongly Inconsistent Correction Subsets (MSICSs).

### B. Minimal Strongly Inconsistent Correction Subsets

A subformula $\phi_s \subseteq \phi$ is strongly inconsistent if $\phi_s$ is inconsistent and all the supersets $\phi'$ of the subformula ($\phi_s \subseteq \phi' \subseteq \phi$) are also inconsistent. Given an *inconsistent* formula $\phi = \phi_h \wedge \phi_s$ where $\phi_h$ is a set of hard constraints and $\phi_s$ is a set of soft constraints, a Minimal Strongly Inconsistent Correction Subset (MSICS) $\phi_c$ is a minimal set of soft constraints ($\phi_c \subseteq \phi_s$) that need to be removed so that the remaining soft and hard constraints are *not strongly inconsistent*. For the purpose of fault localization in an ASP program, an MSICS is a set of lines of the faulty program that must be removed or modified because it is preventing the program from behaving correctly for a given test case.

Consider the faulty student submission for the vertex cover problem presented in the introduction. FormHe found a test case for which this submission fails:

```
#const k = 3.
e(1,2).
```




This test case specifies a vertex cover with a maximum size of three for the simple graph shown. The student implementation fails because it uses `k` as both an upper and lower bound for the size of the cover when it should be just an upper bound. Hence, the student submission does not output any answer set for this test case. One of the missing answer sets is `sel(2)` since selecting just vertex 2 is a cover of the graph. Using this information, we can compute an MSICS of the program.

Consider Figure 3. First, we turn each of the lines in the student program into soft constraints (1). Next, we specify the test case (2), and the missing answer set (3) as hard constraints. From these soft and hard constraints, we can use an MSICS algorithm [11] to compute a minimal set of lines of the program (1) that need to be removed or modified such that the missing answer set (3) can become a solution for the test case (2). More details can be found in the appendix.

Due to the non-monotonicity of ASP, the MSICS approach can sometimes fail to identify all the faulty lines in the program. To overcome this, FormHe can combine the information from the MSICS with supplemental sources of information that can select other suspicious lines. Next, we introduce two alternative fault localization methods.

### C. Large Language Models

Deep Learning models have proven to be a powerful tool for fault localization in different domains [12]–[14]. One way

```
1 v(X) :- e(X,_).
2 v(X) :- e(_,X).
3 k { sel(X) : v(X) } k.
4 :- not sel(X), not sel(Y), e(X,Y).
```

(a) Student submission.

```
1 v(X) :- e(X,_).                                   Soft
2 v(X) :- e(_,X).
3 k { sel(X) : v(X) } k.   (1)
4 :- not sel(X), not sel(Y), e(X,Y).
5 #const k = 3.   (2)                               Hard
6 e(1,2).
7 sel(2).   (3)
```

(b) Relaxed program.

Fig. 3: Student submission for the vertex cover problem before and after relaxation.

to use LLMs for fault localization is to transform the model into a classifier by replacing the last layer with a classification head. This approach allows us to generate a score for each line, indicating the likelihood that the line is faulty. While this approach has the downside of requiring that the maximum number of lines in the program be defined at training time, this is not an issue for the small introductory ASP problems we are targeting.

Our prompt template contains the problem name (e.g., graph k-coloring), the reference implementation and the student submission. The model has been fine-tuned to receive the prompt and output a score between 0 and 1 for each line. Lines with a score $\geq 0.5$ are considered faulty.

### D. Line Matching

The Line Matching method finds lines in the student submission that are very similar to lines in the reference implementation yet have small differences. The intuition is that if such lines exist, they are likely to be bugs. On the contrary, very different lines are likely to be due to entirely different implementation approaches instead.

The first step in Line Matching is parsing both implementations into FormHe's internal Abstract Syntax Tree (AST) representation. Step (A) of Figure 4 shows an example of this parsing step. Then, in step (B), the AST is anonymized, becoming an Anonymized Abstract Syntax Tree (AAST). This is done because students might use different names for auxiliary predicates and variables. Constants and the solution predicates are not anonymized since they would be the same for both the student and the reference implementation.

The last step, (C), turns the AAST into a bag of nodes. This representation is used since many syntax aspects for ASP programs are order-independent (e.g., the order of the atoms in the body of a rule). Furthermore, predicate and variable usages become even more abstract since we cannot rely on the order of the first appearance of each identifier. For example, consider

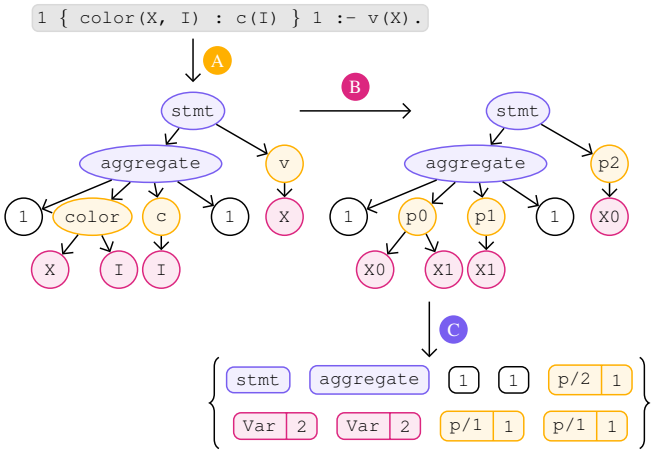



Fig. 4: Process of transforming an ASP rule into a bag of nodes.

the predicate `color`, which is used one time in the original program. At first, it is anonymized into predicate `p0` since it is the first predicate encountered in this line. Then, in the final version, it shows simply as a predicate with two arguments (`p/2`) that is used one time (`p/2 | 1`).

After computing the bag of nodes for all lines in both programs, we compute the symmetric difference for each pair of lines from one program to the other. This metric gives us the nodes that need to be removed and/or added to transform one line into another. Then, we create a bipartite graph between the lines of the submission and the lines of the reference implementation, where the weight of each edge is the previously computed distance metric. Finally, we use a perfect matching algorithm [15] to find a minimum cost pairing between the lines of the submission and reference implementation. Matched lines from the student submission with a small but non-empty symmetric difference (by default $\leq 3$) are reported as likely to be buggy.

### *E. Choosing a correct implementation*

It is possible to build a large set of correct solutions for each exercise by collecting correct student submissions from previous course editions. The student's solutions are potentially different from the reference implementation and can be exploited to improve the quality of the similarity-based fault localization methods [8], namely the LLM approach and the line matching approach. To achieve this, we use the core technique of the line matching algorithm to compute a distance metric (the total value of the matching) between the student submission and each of the available correct implementations for that problem. Then, we select the lowest distance correct implementation and use it as a reference in the similarity-based fault localization methods, improving their performance.

### *F. Combining Fault Localization Methods*

To produce a more robust fault localizer, we combine the different fault localization techniques presented in this section. Figure 5 shows an overview of our combination and sorting

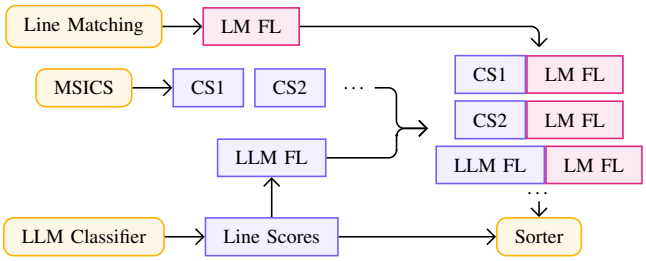



Fig. 5: Overview of how FormHe combines different fault localization techniques.

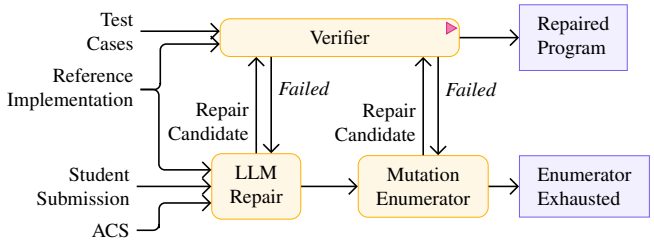



Fig. 6: System diagram of FormHe's repair module. The ▶ symbol represents usage of the Clingo ASP solver.

method. First, we collect all the MSICSs produced (CS1, CS2, ...). Then, we use the LLM fault localization approach to obtain another set of faulty lines (LLM FL). For the rest of the process, this set is treated as if it were an MSICS.

Next, we compute the set of suspicious lines using the Line Matching approach. These lines are appended to all the MSICSs (plus the LLM FL), resulting in Augmented Correction Subsets (ACSs). Finally, we use the Line Scores from the LLM Classifier to sort the ACSs, in order to choose which one to present to the user and proceed with for the repair module. Each ACS is ranked based on the sum of the scores of the lines it contains, with ACSs with higher scores being ranked first.

## IV. Program Repair

The repair module takes the Augmented Correction Subset (ACS) produced by the Fault Localizer and tries to modify it to correct the original program. It is possible that the ACS produced by the fault localizer is empty. This happens when there are no incorrect rules (only missing rules) or when the fault localizer is unable to identify them. If the ACS is empty, FormHe attempts to extend the submitted program with new lines while maintaining the original ones.

Figure 6 shows an overview of the Repair module. FormHe possesses two ways to repair programs: a fine-tuned Large Language Model (LLM) and a program mutation enumerator. First, FormHe attempts to repair the program using the fine-tuned LLM. This is done in a feedback loop where we attempt to refine the answers provided by the LLM until we find a correct repair. After a preset number of iterations of this loop, if no repair has been found, FormHe falls back to the mutation-based repair. In this second phase, the Mutation Enumerator

takes the original student submission and the ACS identified by the fault localizer, and generates mutations of the identified lines. As in the LLM Repair, this starts a feedback loop where different repair candidates are enumerated until we either find one that is correct or the enumerator runs out of possible mutations.

The verification process is similar to assessing a student's submission for correctness. We replace the ACS in the student submission with the repair candidate and check if all answer sets of the resulting program are answer sets of the reference implementation for all test cases. We also ensure that the program generates at least one answer set for each input, mirroring the behavior of the reference implementation. When FormHe identifies a correction, it offers the student a hint in the form of a program with holes where changes were introduced to produce a repair. This hint is more precise than the ACS produced by the fault localizer. A detailed example of how the hints are produced can be found in Section V.

### A. LLM Program Repair

Our LLM repair approach consists of using a fine-tuned LLM to obtain candidate repairs of the ACS found during fault localization. The model was fine-tuned on synthetic data using an input prompt containing the name of the problem, the reference implementation, the student submission and the set of faulty lines (e.g., the ACS). As introduced in the Fault Localization section, similarity-based methods can benefit from having a more closely related correct implementation instead of the reference solution initially produced by the faculty. Hence, in the LLM-based repair, we replace the reference implementation in the input prompt whenever a closer correct implementation is available.

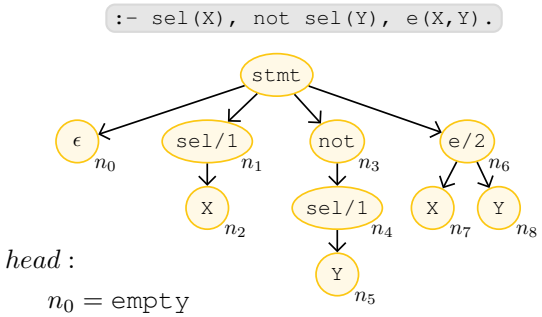
If the repair candidate produced by the LLM is incorrect, FormHe enters a feedback loop where it runs fault localization on that candidate and then tries to further repair it with the LLM again. We do this for a preset number of times (3 by default). If no correction is found, we fall to the mutation-based repair approach.

### B. Mutation Program Repair

The mutation-based repair technique finds repair candidates by enumerating mutations of the ACS. FormHe uses program synthesis to enumerate mutations and, like most enumeration-based program synthesis tools, uses a Domain Specific Language (DSL) to define the repair search space. FormHe's DSL includes the most common ASP operators, such as Boolean operators and aggregate rules. Furthermore, it also takes into account information from the student submission, such as predicate and variable names. Figure 7 shows the base DSL supported by FormHe. The three productions not defined in this figure (*predicate*, *constant*, and *variable*) are dependent on the user submission and on the ACS. The *predicate* production can be any predicate used in the student submission, as well as any of the solution predicates for that instance. FormHe ensures that the arity of each predicate is respected. Likewise, *constant* can be any constant used in the user submission.

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \texttt{stmt}(\mathit{head}?, \mathit{atom}*) \\
\mathit{head} &\rightarrow \texttt{aggregate}(\mathit{term}?, \mathit{atom}, \mathit{atom}, \mathit{term}?) \\
&\quad \mid \mathit{atom} \\
\mathit{atom} &\rightarrow \mathit{predicate}(\mathit{pred\_term}*) \mid \texttt{not}(\mathit{atom}) \\
&\quad \mid \texttt{classical\_not}(\mathit{atom}) \\
&\quad \mid \texttt{eq}(\mathit{term}, \mathit{term}) \mid \texttt{neq}(\mathit{term}, \mathit{term}) \\
&\quad \mid \texttt{lt}(\mathit{term}, \mathit{term}) \mid \texttt{le}(\mathit{term}, \mathit{term}) \\
&\quad \mid \texttt{gt}(\mathit{term}, \mathit{term}) \mid \texttt{ge}(\mathit{term}, \mathit{term}) \\
&\quad \mid \texttt{pool}(\mathit{atom}, \mathit{atom}) \\
\mathit{pred\_term} &\rightarrow \mathit{term} \mid \mathit{interval} \mid \_ \\
\mathit{term} &\rightarrow 0 \mid 1 \mid \mathit{constant} \mid \mathit{variable} \\
&\quad \mid \texttt{add}(\mathit{term}, \mathit{term}) \mid \texttt{sub}(\mathit{term}, \mathit{term}) \\
&\quad \mid \texttt{mul}(\mathit{term}, \mathit{term}) \mid \texttt{div}(\mathit{term}, \mathit{term}) \\
&\quad \mid \texttt{abs}(\mathit{term}) \\
\mathit{interval} &\rightarrow \texttt{interval}(\mathit{term}, \mathit{term})
\end{aligned}
$$

Fig. 7: FormHe's base Domain Specific Language.




$$
\begin{aligned}
&\mathit{head}: \\
&\quad n_0 = \texttt{empty} \\
&\mathit{body}: \\
&\quad n_1 = \texttt{sel/1} \wedge n_2 = \texttt{X} \wedge n_3 = \texttt{not} \wedge n_4 = \texttt{sel/1} \\
&\quad n_5 = \texttt{Y} \wedge n_6 = \texttt{e/2} \wedge n_7 = \texttt{X} \wedge n_8 = \texttt{Y}
\end{aligned}
$$

Fig. 8: An ACS with a single statement, its AST representation, and the SMT encoding for that tree.

Finally, *variable* can be any of the variables used in the ACS as well as completely new variables.

The first step in the mutation-based repair process is to convert the ACS into FormHe's Domain Specific Language and create an AST. Then, the AST is encoded into a Satisfiability Modulo Theories (SMT) formula [16]–[21]. Figure 8 shows an example of an ACS and respective AST and SMT encoding. Relaxing each of the equalities in this formula allows us to use an SMT solver to enumerate program mutations. For each node $n_i$ we introduce a relaxation variable $r_i$. For example, relaxing $n_2 = \texttt{X}$ would become $(n_2 = \texttt{X} \vee r_2)$ and allow us to enumerate `:- sel(Y), not sel(Y), e(X,Y).`.

To reduce the number of enumerated programs that are either semantically incorrect ASP programs or that are mathematically equivalent, we enforce several restrictions that prune enumerated programs. These include symmetry breaking

for commutative operations, ignoring neutral and absorbing elements for addition and multiplication, and several other optimizations related to ASP semantics, such as pruning invalid constructions of rule heads.

## V. FormHe in the Classroom

In the last two years, master's-level Automated Reasoning class students have submitted their exercise solutions for ASP programs. However, the only feedback was checking if the submitted programs passed all test cases. No hints or personalized feedback was provided. This year, FormHe using the techniques described in this paper will be tested by students for the first time. In this section, we describe and show an example of how users interact with FormHe in the classroom.

Students interact with FormHe through GitSEED [22], an open-source tool that allows teachers to set up custom automatic feedback on Git commits. We create an initial repository for each student containing each exercise's statement and skeleton file. GitSEED will then watch the repository for commits on the skeleton files and trigger FormHe whenever one happens.

Consider as an example the following program committed by a student as a solution for the $k$-coloring problem:

```
1 color(1..k).
2 node(N) :- e(_, N).
3 node(N) :- e(N, _).
4 assign(N,C) :- node(N), color(C).
5 :- e(N, M), assign(N,C), assign(M,C).
```

After being triggered by GitSEED, FormHe evaluates the student program against the test cases. For each failing test case, we show the student the input as well as some of the missing/extra answer sets. The following block shows the evaluation output for the student program:

> Your program has failed test case 1:
>
> ```
> #const k = 3.
> e(a, b). e(a, c). e(a, d). e(c, d).
> ```
>
> Your solution is overconstrained and does not produce any solutions for this input. Examples of correct answer sets:
>
> ```
> assign(a,3) assign(b,1) assign(c,1) assign(d,2)
> assign(a,3) assign(b,2) assign(c,1) assign(d,2)
> assign(a,3) assign(b,2) assign(c,2) assign(d,1)
> assign(a,3) assign(b,1) assign(c,2) assign(d,1)
> assign(a,2) assign(b,3) assign(c,3) assign(d,1)
> ```
>
> **[tests 2 to 4 omitted for brevity]**
>
> You have passed 0 test(s) and failed 4 test(s).

Additionally, FormHe can detect if the student has used a predicate in the body of some rule that does not occur in any head (such predicate will never be satisfied and is usually a mistake). For example, if that happens for predicate `my_predicate/2`, the following line is also added to the output:

> The following predicates are used in a rule but are never generated: my_predicate/2

Next, FormHe executes the fault localization procedure and, if any lines are identified, produces the following information:

> Suggested lines where bug is likely located:
>
> **[4]** `assign(N, C) :- node(N), color(C).`

At this point, while FormHe starts the repair procedure, GitSEED will push the fault localization output to the student's Git repository in a Markdown file, along with a message indicating that FormHe is searching for a correction hint and will update the file shortly. When the repair procedure terminates, FormHe will create a correction hint. Alternatively, if the timeout is reached, it informs that no hint was found. GitSEED then updates the feedback file on the students' repository. In this example, the student incorrectly used a regular rule when they needed an aggregate rule. FormHe successfully identified the repair `1 { assign(N, C) : color(C) } 1 :- node(N).` and generated the following fix suggestion:

> **Fix Suggestion**
>
> You can try replacing the line(s) above with the following (the "?" are holes you should fill in or remove):
>
> ```
> ? :- node(N), ?.
> ```

As shown in this section, FormHe has several types of feedback with different characteristics:

- Information about failing test cases is always available but can be hard to utilize effectively by students with little experience;
- Fault localization information is more useful but is only available if FormHe can identify the bugs;
- Fix suggestions are more precise but depend on finding a correction for the program and often take longer to obtain.

FormHe uses GitSEED to make interaction with users as simple as possible and provides feedback as it becomes available.

## VI. Evaluation

To properly evaluate FormHe's capabilities, we used programs from students taking a master-level class on Automated Reasoning. Over two years, we collected 115 programs submitted by students on 5 different assignments. In these assignments, students encoded classical graph problems such as graph coloring and vertex cover, as well as set problems such as pairwise disjoint sets and set cover. Of those 115 instances, we received 63 correct submissions and 52 semantically incorrect submissions.

To better evaluate our tool, we also created synthetic benchmarks. These benchmarks were generated by randomly introducing 1 to 8 mutations in the correct submissions, including some of the most common bug types, such as using a wrong predicate name or forgetting a constraint. We created 95k

instances for training the different machine-learning models and 500 instances for evaluation.

This section answers the following research questions:

**Q1.** How effective are the fault localization approaches?

**Q2.** Can we improve the fault localization through a combination of approaches?

**Q3.** How effective is the program repair?

We evaluated our tool using an Intel Xeon Silver 4210R and imposed a limit of 10 minutes (wall clock time) and 60GB of RAM per instance. Limits were strictly imposed using Runsolver [23]. FormHe is implemented in Python and uses the Clingo ASP grounder and solver [24] version 5.6.2. For the enumeration of program mutations, FormHe uses a modified version of the Trinity framework [16]. Fine-tuning and evaluation of the different LLMs was performed using 5x Nvidia RTX A4000. FormHe's source code, data and logs are available as supplemental material.

*Q1: How effective are the different fault localization approaches?*

We start by evaluating the fault localization approaches in FormHe. Note that while the different fault localization approaches are intended to be used together, they can also be used in isolation. Table I shows the percentage of instances where each approach correctly identified the faults for real and synthetic instances.

*a) MSICS Method:* The MSICS fault localization method has the largest number of Exact Faults Identified out of the two traditional methods. However, due to the non-monotonicity of ASP, this method produces some unexpected results, with a large number of non-identified faults and wrong identifications. For instance, when the body of a rule has a bug and is never satisfied, that rule does not contribute to the logical behavior of the program and will thus not be identified by this method. Even so, Table I shows that this method correctly identifies at least one fault in a large number of instances (64%) and provides a good baseline.

*b) Line Matching:* The line matching algorithm exploits similarities between the student submission and the reference implementation. While this approach may be less helpful for complex programs with many ways to solve the problem, it performs well for simple instances such as those used in introductory ASP classes. The large number of wrong identifications for synthetic instances shown in Table I is due to these instances using a normalized representation which can sometimes cause false positives with the reference implementation (for example, `0 {...} 1` is syntactically different but equivalent to `{...} 1`).

*c) Large Language Models:* We use open-access LLMs with a small number of parameters for three reasons: (1) closed-access models are prohibitive due to cost and students' data privacy concerns, (2) models with a large number of parameters require large amounts of computing power and take longer to produce answers, which is undesirable in a classroom, and (3) since we transform the models into classifiers with few outputs, too many parameters can be detrimental and lead to overfitting. Furthermore, we use fine-tuned models due to two major issues: (1) many models have trouble reasoning about ASP, likely due to a small proportion of this language in their training sets, and (2) some models have trouble respecting specific output formats which makes it hard to automatically extract the relevant parts of the answer for verification and/or use in other FormHe modules. These issues can be partially alleviated by using large model sizes, but, as explained, that is undesirable for FormHe's intended use.

We fine-tuned four state-of-the-art models to evaluate our LLM fault localization approach. We chose two models with 2B parameters: Gemma [25] and CodeGemma [26], a model with 3B: StarCoder 2 [27], and a model with 4B: Phi 3 [28]. These models were fine-tuned using 95k synthetically generated incorrect programs. We used Parameter Efficient Fine Tuning [29] (in particular Low-Rank Adaptation (LoRa) [30]) to decrease the memory requirements during training.

Table I shows the results for different models for real and synthetic instances. Note that the synthetic results are for the 500 evaluation instances and not for the 95k used for training. Of the four models, Gemma performed the best overall. While Phi 3 performs better than Gemma for synthetic instances, the same is not true for real instances. This indicates that Phi 3 has likely overfitted the training data. Furthermore, the overall disparity between the results for synthetic and real instances is expected since the models were also fine-tuned using synthetic instances. Even with these caveats, the results for real instances are generally better than the other fault localization approaches.

*d) Related Methods:* To compare FormHe with previous approaches, we also implemented the core DWASP [2] fault localization method (excluding the interactive portion) in our tool. The results for this method can be found as DWASP$^\dagger$ in Table I. The DWASP$^\dagger$ approach is very similar to the MSICS method, and this shows in the results, with comparable performance for these approaches. Even though the DWASP$^\dagger$ slightly outperforms the MSICS method in real instances, replacing it in FormHe's default configuration results in worse performance, suggesting that DWASP$^\dagger$ has a greater overlap with the other methods.

*Q2: Can we improve the fault localization through a combination of approaches?*

Table I shows the results for a configuration of FormHe without using LLMs ("FormHe w/o LLM") and for the recommended configuration using the Gemma 2B model ("FormHe (with Gemma 2B)"). Combining the MSICS and Line Matching methods provides a significant performance improvement compared to the isolated approaches. Furthermore, in deployment scenarios capable of using deep learning, adding the LLM fault localization method provides a further boost, with the recommended configuration finding all faults in $85\%$ of real submissions and at least one of the faulty lines in $94\%$. Even though FormHe has a much larger number of "Superset Faults Identified" cases than other approaches, for 78% of those cases, only 1 extra line was identified. This means that

TABLE I: Results for different fault localization methods for real and synthetic instances. Label meanings: "Exact Faults Identified" — the method identified all the faulty lines and no others; "Superset Faults Identified" — the method identified all the faulty lines, but also some others; "Some Faults Identified" — the method identified some of the faults, but not all; "Faults Not Identified" — the program had faulty lines but the method was unable to identify them; "Wrong Identification" — the method only identified lines as faulty that were actually correct. The first column shows the sum of the Exact, Superset, and Some Faults Identified columns and represents the percentage of instances where at least one fault was found in the program.

| | Exact + Superset + Some Faults Identified | | Exact Faults Identified | | Superset Faults Identified | | Some Faults Identified | | Fault Not Identified | | Wrong Identification | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Real | Synth. | Real | Synth. | Real | Synth. | Real | Synth. | Real | Synth. | Real | Synth. |
| MSICS | 63.5% | 46.0% | 40.4% | 24.2% | 0.0% | 0.4% | 23.1% | 21.4% | 23.1% | 42.6% | 13.5% | 11.4% |
| Line Matching | 69.2% | 48.2% | 34.6% | 20.2% | 11.5% | 3.8% | 23.1% | 24.2% | 25.0% | 40.4% | 5.8% | 11.4% |
| Gemma 2B | 90.4% | **100.0%** | 59.6% | 99.6% | 17.3% | 0.0% | 13.5% | 0.4% | 5.8% | 0.0% | 3.8% | 0.0% |
| CodeGemma 2B | 84.6% | **100.0%** | 50.0% | 99.2% | 19.2% | 0.0% | 15.4% | 0.8% | 9.6% | 0.0% | 5.8% | 0.0% |
| StarCoder2 3B | 65.4% | 99.2% | 26.9% | 96.6% | 7.7% | 0.2% | 30.8% | 2.4% | 26.9% | 0.6% | 7.7% | 0.2% |
| Phi 3 mini | 84.6% | **100.0%** | 51.9% | 100.0% | 13.5% | 0.0% | 19.2% | 0.0% | 9.6% | 0.0% | 5.8% | 0.0% |
| FormHe w/o LLM | 82.7% | 69.8% | 44.2% | 32.2% | 13.5% | 8.8% | 25.0% | 28.8% | 11.5% | 19.8% | 5.8% | 10.4% |
| → **FormHe** (with Gemma 2B) | **94.2%** | 97.8% | 61.5% | 77.6% | 23.1% | 17.8% | 9.6% | 2.4% | 1.9% | 0.0% | 3.8% | 2.2% |
| FormHe w/o Impl. Choosing | 92.4% | 98.0% | 63.5% | 71.4% | 21.2% | 25.8% | 7.7% | 0.8% | 1.9% | 0.0% | 5.8% | 2.0% |
| DWASP† | 69.3% | 31.0% | 46.2% | 19.6% | 1.9% | 0.6% | 21.2% | 10.8% | 30.8% | 62.8% | 0.0% | 6.2% |

FormHe is still helping the student focus on the problematic section of the program.

We also explored the impact of using the closest correct implementation for the Line Matching and LLM methods instead of the reference implementation. For a given instance, we only consider previously submitted correct solutions, simulating real-time usage of FormHe. The line labeled "FormHe w/o Impl. Choosing" shows the effects of disabling this feature compared with the default configuration of FormHe. Disabling the feature has a small impact, decreasing the number of instances in which we find at least one fault and raising the number of Wrong Identifications.

Our combined approach can successfully incorporate the different fault localization methods, obtaining the best overall results of all techniques for real instances. FormHe can provide students actionable feedback for the majority of faulty submissions. Furthermore, even for submissions where we cannot identify the fault, we can still inform the student that the program is incorrect and provide failing test cases with missing and extra answer sets.

*Q3: How effective is the program repair?*

Next, we evaluate the performance of the repair module using the best-performing fault localization method (FormHe with Gemma 2B). Figure 9 shows how many real instances can be repaired under $x$ seconds for different configurations of FormHe. Times shown in this plot include the time for the fault localizer (5 seconds on average). The model used for the LLM and Combined configurations was CodeGemma 7B fine-tuned using LoRa and 4 bit quantization. We used a larger number of parameters for repair than for fault localization since the repair task is more complex.

The "Mutation Repair" and "LLM Repair lines" refer to using each of the two repair techniques in isolation. The LLM approach has a much better repair rate at 56% compared to

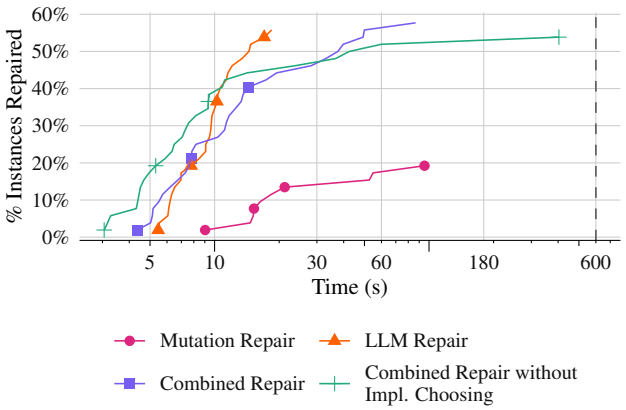



Fig. 9: Percentage of submissions (i.e., real instances) repaired at each point in time.

19% for the mutation-based. In part, this happens because the mutation method's default configuration has been tuned towards harder instances. While this reduces the overall number of mutation-based repairs, it increases the contribution to the combined approach described next.

The combined approach slightly improves over using just the LLM repair, with 58% repaired instances. Even though mutation-based repair makes a small contribution to the combined approach, it is not dependent on having synthetic instances and performing computationally intensive fine-tuning of models. It can thus be used in low-resource situations and provides a safe fallback when the LLM fails. Observe that almost all repairs are found in 1 minute or less. This is ideal for classroom situations since students should not wait a long time for feedback.

The Combined Repair configuration uses the closest correct implementation available in the LLM input prompt. The impact

of disabling this feature is shown in Figure 9 as "Combined Repair without Impl. Choosing". This feature has a larger impact on repair performance than on fault localization with a 4pp. drop in repair rate when it is disabled. There is also a large impact on the repair time. Using the closest solution takes at most about 2 minutes versus 8 minutes when using the original reference solution.

Although not shown in the figure, synthetic instances show the same behavior as in fault localization: the LLM approach performs much better at synthetic instances than real instances ($97\%$ vs $56\%$), while the traditional method (mutation-based) performs similarly on the two types of instances ($20\%$ vs $19\%$). This can be explained by the LLM having been trained on synthetic instances (although not the same ones as used for evaluation). We also fine-tuned other models for the repair task besides CodeGemma 7B. These are: 8-bit quantized CodeQwen 1.5 7B [31] with $48\%$ combined repair rate for real instances, Phi 3 mini with $40\%$ and CodeGemma 2B with $37\%$.

FormHe's repair rate depends significantly on the fault localization outcome. Looking at all 552 instances (real + synthetic), FormHe can repair 96% of the ones with Exact Faults Identified, 92% of Superset Faults Identified and 71% of Some Faults Identified. This shows that better fault localization leads to better repair rates and also that it is preferable to identify a superset of the faulty lines than to only identify some of the faults. Furthermore, FormHe can also repair 50% of the instances where the faults were Not Identified or Wrongly Identified. This occurs in instances where the faults are missing ASP rules. Although the fault localization module does not always detect this, the repair module can sometimes solve the problem by synthesizing new rules.

## VII. Related Work

### A. Verification and Similarity of ASP

While several techniques for formal equivalence checking of Answer Set Programs have been proposed [32]–[34], FormHe uses testing-based verification for two main reasons: (1) many of these techniques do not directly support all ASP features (e.g., aggregates) and require program transformations and/or grounding which increase complexity and make giving feedback harder, and (2) FormHe calls the verification procedure very often to test if the candidate repairs are correct or not, so it needs to be as fast as possible. Furthermore, since the faculty write the tests, they can ensure the inputs expose all common corner cases.

Techniques using similarity of ASP rules have previously been used for plagiarism detection in Kato [35]. Kato's goal is to detect camouflage techniques such as changing the formatting of the source code or permuting rules/atoms in the program. While FormHe's line matching and Kato share underlying ideas, our goal is actually the opposite: we want to ignore any "irrelevant" differences between programs and find semantic bugs.

### B. Fault Localization

Work on fault localization for ASP has been ongoing. Shchekotykhin [1] presents an interactive tool for ASP fault localization that iteratively asks the user if a given set of atoms should be part of some answer set. Based on this information, the tool refines the set of possibly faulty rules until it finds the correct one. Users of this tool must be able to reason about the semantics of ASP, which makes it challenging to use in an education setting where students are just learning declarative languages.

SeaLion [3] is an ASP debugger which allows users to compute answer sets step-by-step. Initially, no rules are active, and the answer set is empty. Then, at each step, users can select a new rule to be active, producing changes in the current answer set. This allows users to determine when the answer set deviates from what is expected. This tool requires some intuition to be used effectively since users must choose the correct rules at each step and know when the computation has "gone wrong". SeaLion also includes Ouroboros [36], [37], a tool which allows users to compute why a given interpretation is not an answer set of the program by using a meta interpreter. In this plugin, users must choose an interpretation that correctly exposes the bug, which can be challenging if the program is overconstrained instead of underconstrained.

DWASP [2] is a debugger based on the WASP solver [38] that requires an input (e.g., an example graph for the vertex cover problem), and a test case (a set of atoms that should be part of some answer set but are not). DWASP then computes a minimal set of lines that make the test case incoherent with the input. Although the *reason of incoherence* is minimal from a logic standpoint, it usually includes lines that are not faulty. As such, DWASP implements an interactive debugger that allows users to refine the reason for incoherence. Although this approach is more automated than previous work, it still requires users to (1) select a good input and test case that exposes the fault and (2) refine the set of possible lines by answering questions that require complex reasoning.

### C. Deep Learning Fault Localization

Recently, many approaches to fault localization using deep learning have been proposed. DeepFL [12] uses a deep learning model to combine the results of many fault localization techniques (such as mutation-based, textual-similarity, among others) into a single suspiciousness score for each line.

TRANSFER [13] uses bidirectional LSTM-based classifiers to compute deep semantic features of the buggy programs. Then, it uses a different model to combine these features with spectrum and mutation-based metrics and produce suspiciousness scores for each program line.

LLMAO [14] uses a two-step approach where a pre-trained LLM is used to obtain a representation for each program line (the model's hidden state). Then, a bidirectional transformer model is used to transform the sequence of representations into a suspiciousness score for each line. This two-step approach is used to avoid the LLM having to "remember" the whole

program in the hidden state of the last token, which could make it challenging to use with very large programs.

### D. Program Repair

Automated Program Repair (APR) encompasses techniques aimed at automatically fixing bugs in programs, spanning from imperative [39], [40] to declarative paradigms [41]–[43]. To the best of our knowledge, FormHe is the first automated program repair tool for ASP. However, work has been ongoing in other declarative languages, such as Alloy.

ARepair [41] and ICEBAR [42] are two repair tools for Alloy that use AlloyFL as a fault localizer. ARepair uses a sketch-based approach: based on the suspiciousness scores computed by AlloyFL, ARepair selects suspicious nodes and replaces them with holes. Then, a synthesizer tries to fill those holes to produce a correct program. ICEBAR builds upon ARepair by introducing an extra form of specification: a property-based oracle that validates if the model respects some property. Using this oracle, ICEBAR iteratively increases the set of test cases used by ARepair, improving the quality of the repairs and reducing overfitting.

ATR [43] is a repair tool for Alloy that uses FLACK as a fault localizer. ATR uses pairs of counterexamples and closely related satisfying instances. Based on these pairs, ATR constructs candidate repairs in a bottom-up manner. Finally, ATR tries to replace the suspicious statements produced by FLACK with these candidates.

## VIII. Conclusion

This paper proposes FormHe, a fault localization and program repair tool for ASP that combines logic techniques with machine learning. FormHe helps students who are using declarative languages for the first time and do not have the intuition necessary to use other forms of debugging. FormHe assists them in finding and correcting faults in their programs by providing automatic and personalized feedback. FormHe can offer valuable insights to students by correctly identifying faults in 94% of incorrect submissions and providing repair hints in 56% of cases. Furthermore, even when the fault localization and repair are unsuccessful, FormHe can still provide students with a failing example that points them in the right direction.

All techniques proposed in the paper will be deployed in a real scenario in this year's classes. Preliminary results presented in this paper show that integrating logic-based methods and Large Language Models (LLMs) can boost automated fault localization and program repair in declarative programming. Moreover, the strengths of both logic-based methods and LLMs are complementary and improve upon using just one approach.

## Appendix

### A. Computing MSICSs

Consider again the program with hard and soft clauses presented in Figure 3b. Soft clauses are implemented with the help of relaxation variables and an auxiliary aggregate rule. The program below shows the full program after the relaxation step (new parts highlighted in bold):

```
<|problem|>{Problem Title}
<|reference_program|>{Reference Program}
<|incorrect_program|>{Incorrect Program}
```

(a) Fault localization prompt.

```
<|problem|>{Problem Title}
<|reference_program|>{Reference Program}
<|incorrect_program|>{Incorrect Program}
<|fl|>{ACS Lines}
<|missing_lines|>{Yes or No}
<|correction|>
```

(b) Repair prompt.

Fig. 10: LLM input prompts for the fault localization and repair modules.

```
1 v(X) :- e(X,_), _r0.
2 v(X) :- e(_,X), _r1.
3 k { sel(X) : v(X) } k :- _r2.
4 :- not sel(X), not sel(Y), e(X,Y), _r3.
5 #const k = 3.
6 e(1,2).
7 sel(2).
8 0 { _r0 ; _r1 ; _r2 ; _r3 } 4.
```

In this program, each of the first 4 rules is only active when the respective relaxation variable is derived. By iteratively increasing the lower bound on the auxiliary aggregate rule, we can find a minimal set of rules that need to be removed or modified so that the failing example can become correct. More details on the technique for computing MSICSs can be found in the paper by Mencía and Marques-Silva [11].

### B. Fine-tuning Methodology

Figures 10a and 10b show the prompts used for finetuning the fault localization and repair models, respectively. In these prompts, the strings $<|...|>$ each refer to a custom token added to the models' embeddings. This improves ease of training and sligthly decreases the LLM's response time (by decreasing the total number of tokens). For the finetuning of our models, we used 4 epochs, a learning rate of $10^{-4}$, batch size of 1 to 8 (depending on VRAM requirements), gradient accumulation, a LoRA $r$ value of 8, LoRA alpha of 8 and LoRA dropout of 0.05.

The fault localization model was trained using Multi Label Sequence Classification, where each label X from 1 to `MAX_LINES` represents if line number X is faulty or not. Although not mentioned in the main paper, we also include an extra label representing if the program has missing lines.

The repair model was trained using Supervised Finetuning. Besides the inputs mentioned in the main paper, the repair prompt also receives information about if the program has missing lines or not. This information comes directly from the extra label in the fault localization model.

## References

[1] K. M. Shchekotykhin, "Interactive query-based debugging of ASP programs," in *Proceedings of the Twenty-Ninth AAAI Conference*

*on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 1597–1603. [Online]. Available: https://doi.org/10.1609/aaai.v29i1.9394

[2] C. Dodaro, P. Gasteiger, K. Reale, F. Ricca, and K. Schekotihin, "Debugging non-ground ASP programs: Technique and graphical tools," *Theory Pract. Log. Program.*, vol. 19, no. 2, pp. 290–316, 2019. [Online]. Available: https://doi.org/10.1017/S1471068418000492

[3] J. Oetsch, J. Pührer, and H. Tompits, "Stepwise debugging of answer-set programs," *Theory Pract. Log. Program.*, vol. 18, no. 1, pp. 30–80, 2018. [Online]. Available: https://doi.org/10.1017/S1471068417000217

[4] G. Brewka, T. Eiter, and M. Truszczynski, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011. [Online]. Available: https://doi.org/10.1145/2043174.2043195

[5] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[6] D. Maier, K. T. Tekle, M. Kifer, and D. S. Warren, "Datalog: concepts, history, and outlook," in *Declarative Logic Programming: Theory, Systems, and Applications*, M. Kifer and Y. A. Liu, Eds. ACM / Morgan & Claypool, 2018, pp. 3–100. [Online]. Available: https://doi.org/10.1145/3191315.3191317

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. [Online]. Available: https://doi.org/10.2200/S00457ED1V01Y201211AIM019

[8] S. Gulwani, I. Radicek, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 465–480. [Online]. Available: https://doi.org/10.1145/3192366.3192387

[9] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 74:1–74:31, 2022. [Online]. Available: https://doi.org/10.1145/3510418

[10] C. Drescher, O. Tifrea, and T. Walsh, "Symmetry-breaking answer set solving," *AI Commun.*, vol. 24, no. 2, pp. 177–194, 2011. [Online]. Available: https://doi.org/10.3233/AIC-2011-0495

[11] C. Mencía and J. Marques-Silva, "Reasoning about strong inconsistency in ASP," in *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, ser. Lecture Notes in Computer Science, L. Pulina and M. Seidl, Eds., vol. 12178. Springer, 2020, pp. 332–342. [Online]. Available: https://doi.org/10.1007/978-3-030-51825-7_24

[12] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[13] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1169–1180. [Online]. Available: https://doi.org/10.1145/3510003.3510147

[14] A. Z. H. Yang, C. L. Goues, R. Martins, and V. J. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 17:1–17:12. [Online]. Available: https://doi.org/10.1145/3597503.3623342

[15] D. F. Crouse, "On implementing 2d rectangular assignment algorithms," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 52, no. 4, pp. 1679–1696, 2016. [Online]. Available: https://doi.org/10.1109/TAES.2016.140952

[16] R. Martins, J. Chen, Y. Chen, Y. Feng, and I. Dillig, "Trinity: An extensible synthesis framework for data science," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 1914–1917, 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352098

[17] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. Manquinho, "SQUARES : A SQL synthesizer using query reverse engineering," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2853–2856, 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415492

[18] D. Ramos, J. Pereira, I. Lynce, V. Manquinho, and R. Martins, "UNCHARTIT: an interactive framework for program recovery from charts," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 175–186. [Online]. Available: https://doi.org/10.1145/3324884.3416613

[19] A. Ni, D. Ramos, A. Z. H. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "SOAR: A synthesis approach for data science API refactoring," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 112–124. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00023

[20] R. Brancas, M. Terra-Neves, M. Ventura, V. Manquinho, and R. Martins, "Towards reliable SQL synthesis: Fuzzing-based evaluation and disambiguation," in *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, ser. Lecture Notes in Computer Science, D. Beyer and A. Cavalcanti, Eds., vol. 14573. Springer, 2024, pp. 232–254. [Online]. Available: https://doi.org/10.1007/978-3-031-57259-3_11

[21] M. Ferreira, R. Ware, Y. Kothari, I. Lynce, R. Martins, A. Narayan, and J. Sherry, "Reverse-engineering congestion control algorithm behavior," in *Proceedings of the 2024 ACM on Internet Measurement Conference, IMC 2024, Madrid, Spain, November 4-6, 2024*, N. Vallina-Rodriguez, G. Suarez-Tangil, D. Levin, and C. Pelsser, Eds. ACM, 2024, pp. 401–414. [Online]. Available: https://doi.org/10.1145/3646547.3688443

[22] P. Orvalho, M. Janota, and V. Manquinho, "GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education," in *Proceedings of the 2024 ACM Virtual Global Computing Education Conference, SIGCSE Virtual 2024*, vol. 1, 2024. [Online]. Available: https://doi.org/10.1145/3649165.3690106

[23] O. Roussel, "Controlling a solver execution with the runsolver tool," *J. Satisf. Boolean Model. Comput.*, vol. 7, no. 4, pp. 139–144, 2011. [Online]. Available: https://doi.org/10.3233/sat190083

[24] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo," *Theory Pract. Log. Program.*, vol. 19, no. 1, pp. 27–82, 2019. [Online]. Available: https://doi.org/10.1017/S1471068418000054

[25] T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney, I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, and et al., "Gemma: Open models based on gemini research and technology," *CoRR*, vol. abs/2403.08295, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08295

[26] H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman, "Codegemma: Open code models based on gemma," *CoRR*, vol. abs/2406.11409, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.11409

[27] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, and et al., "Starcoder 2 and the stack v2: The next generation," *CoRR*, vol. abs/2402.19173, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.19173

[28] M. I. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. S. Behl, A. Benhaim, M. Bilenko, J. Bjorck, S. Bubeck, M. Cai, C. C. T. Mendes, W. Chen, V. Chaudhary, P. Chopra, A. D. Giorno, G. de Rosa, M. Dixon, R. Eldan, D. Iter, A. Garg, A. Goswami, S. Gunasekar, E. Haider, J. Hao, R. J. Hewett, J. Huynh, M. Javaheripi, X. Jin, P. Kauffmann, N. Karampatziakis, D. Kim, M. Khademi, L. Kurilenko, J. R. Lee, Y. T. Lee, Y. Li, C. Liang, W. Liu, E. Lin, Z. Lin, P. Madan, A. Mitra,

H. Modi, A. Nguyen, B. Norick, B. Patra, D. Perez-Becker, T. Portet, R. Pryzant, H. Qin, M. Radmilac, C. Rosset, S. Roy, O. Ruwase, O. Saarikivi, A. Saied, A. Salim, M. Santacroce, S. Shah, N. Shang, H. Sharma, X. Song, M. Tanaka, X. Wang, R. Ward, G. Wang, P. Witte, M. Wyatt, C. Xu, J. Xu, S. Yadav, F. Yang, Z. Yang, D. Yu, C. Zhang, C. Zhang, J. Zhang, L. L. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, and X. Zhou, "Phi-3 technical report: A highly capable language model locally on your phone," *CoRR*, vol. abs/2404.14219, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.14219

[29] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan, "Peft: State-of-the-art parameter-efficient fine-tuning methods," https://github.com/huggingface/peft, 2022.

[30] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9

[31] Qwen Team, "Code with codeqwen1.5," April 2024. [Online]. Available: https://qwenlm.github.io/blog/codeqwen1.5/

[32] T. Eiter and M. Fink, "Uniform equivalence of logic programs under the stable model semantics," in *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, ser. Lecture Notes in Computer Science, C. Palamidessi, Ed., vol. 2916. Springer, 2003, pp. 224–238. [Online]. Available: https://doi.org/10.1007/978-3-540-24599-5_16

[33] T. Eiter, H. Tompits, and S. Woltran, "On solution correspondences in answer-set programming," in *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, 2005, pp. 97–102. [Online]. Available: http://ijcai.org/Proceedings/05/Papers/1177.pdf

[34] J. Oetsch, M. Seidl, H. Tompits, and S. Woltran, "Beyond uniform equivalence between answer-set programs," *ACM Trans. Comput. Log.*, vol. 22, no. 1, pp. 2:1–2:46, 2021. [Online]. Available: https://doi.org/10.1145/3422361

[35] J. Oetsch, J. Pührer, M. Schwengerer, and H. Tompits, "The system kato: Detecting cases of plagiarism for answer-set programs," *Theory Pract. Log. Program.*, vol. 10, no. 4-6, pp. 759–775, 2010. [Online]. Available: https://doi.org/10.1017/S1471068410000402

[36] J. Oetsch, J. Pührer, and H. Tompits, "Catching the ouroboros: On debugging non-ground answer-set programs," *Theory Pract. Log. Program.*, vol. 10, no. 4-6, pp. 513–529, 2010. [Online]. Available: https://doi.org/10.1017/S1471068410000256

[37] M. Frühstück, J. Pührer, and G. Friedrich, "Debugging answer-set programs with ouroboros - extending the sealion plugin," in *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, P. Cabalar and T. C. Son, Eds., vol. 8148. Springer, 2013, pp. 323–328. [Online]. Available: https://doi.org/10.1007/978-3-642-40564-8_32

[38] M. Alviano, C. Dodaro, N. Leone, and F. Ricca, "Advances in WASP," in *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, ser. Lecture Notes in Computer Science, F. Calimeri, G. Ianni, and M. Truszczynski, Eds., vol. 9345. Springer, 2015, pp. 40–54. [Online]. Available: https://doi.org/10.1007/978-3-319-23264-5_5

[39] D. Ramos, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Batfix: Repairing language model-based transpilation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, p. 161, 2024. [Online]. Available: https://doi.org/10.1145/3658668

[40] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.104

[41] K. Wang, A. Sullivan, and S. Khurshid, "Automated model repair for alloy," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 577–588. [Online]. Available: https://doi.org/10.1145/3238147.3238162

[42] S. G. Brida, G. Regis, G. Zheng, H. Bagheri, T. Nguyen, N. Aguirre, and M. F. Frias, "ICEBAR: feedback-driven iterative repair of alloy specifications," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 55:1–55:13. [Online]. Available: https://doi.org/10.1145/3551349.3556944

[43] G. Zheng, T. Nguyen, S. G. Brida, G. Regis, N. Aguirre, M. F. Frias, and H. Bagheri, "ATR: template-based repair for alloy specifications," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 666–677. [Online]. Available: https://doi.org/10.1145/3533767.3534369