TPLP: Page 1–16.
doi:10.1017/S1471068426100507

ProDebug: *An Automated Debugging System for Prolog*[*]

RICARDO BRANCAS and VASCO MANQUINHO
Universidade de Lisboa Instituto Superior Técnico, Portugal
INESC-ID, Portugal
(*e-mails:* ricardo.brancas@tecnico.ulisboa.pt, vasco.manquinho@inesc-id.pt)

RUBEN MARTINS
Carnegie Mellon University, USA
(*e-mail:* rubenm@andrew.cmu.edu)

submitted 26 May 2026 UTC; revised 26 May 2026 UTC; accepted 28 May 2026 UTC

Abstract

Prolog is a well-known declarative programming language commonly used in introductory courses on logic and reasoning. However, many students find Prolog challenging because it lacks the familiar debugging mechanisms found in imperative languages. In large classes, this difficulty is exacerbated by the challenge of providing timely and personalized feedback to students. In this work, we introduce ProDebug, the first tool to combine large language models (LLMs) with spectrum-based and mutation-based techniques for automated debugging of Prolog assignments. ProDebug automatically identifies faults and proposes bug repairs for student Git submissions. Faults are detected using three approaches – spectrum-based, mutation-based, and LLM reasoning – while repairs are generated using mutation-based techniques and LLMs. Our evaluation on 1499 buggy student submissions from a bachelor's level programming class demonstrates the potential of automated, LLM-augmented feedback systems to scale support for declarative programming education.

KEYWORDS: declarative programming education, automated fault localization and repair, large language models

1 Introduction

Debugging logic programs in Prolog poses a unique set of challenges. Traditional methods like tracing or inserting print statements are often insufficient, since Prolog's execution

[*] This work was supported by Portuguese national funds through FCT, under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), 2023.14280.PEX (DOI: https://doi.org/10.54499/2023.14280.PEX) and through the Carnegie Mellon University Portugal Program under grant PRT/BD/152086/2021 (DOI: https://doi.org/10.54499/PRT/BD/152086/2021). This work was also partially supported by the National Science Foundation (NSF) under Award CCF2427581 and DARPA under Agreement FA8750-24-9-1000.

model is based on unification and backtracking rather than explicit control and data flows. This difficulty is particularly evident among learners, who often struggle to grasp the seemingly unpredictable flow of execution. While some debugging aids for Prolog have been proposed in the past (Neumerkel 1996; Le and Pinkwart 2011; Thompson and Sullivan 2020), they frequently rely on inflexible rule-based approaches and have cumbersome methods of interaction for students. As a result, many students rely on trial and error to detect and fix mistakes, making the learning process in Prolog and declarative programming more frustrating. To address this challenge, we introduce a tool that automatically analyzes Prolog programs, identifies possible bugs, and provides hints to help students correct their solutions.

Consider the task of defining a predicate `duplicate/2` that duplicates every element of a list. For instance, the query `?- duplicate([1,2],[1,1,2,2]).` should succeed. Below is a student's submission for this problem.

```
1 duplicate([], []).
2 duplicate([H|T], L2) :- duplicate(T, L1), L2 = (H,H,L1).
```

While the base case is correct, the recursive case is flawed. Instead of constructing a list, the student mistakenly creates a term of the form `(H,H,L1)`, which does not represent the intended Prolog list structure. Our tool aims to automatically identify such errors and provide students with targeted hints. For this case, a possible hint, where `?` highlights the part of the program the student should revise, could be:

```
1 duplicate([], []).
2 duplicate([H|T], L2) :- duplicate(T, L1), L2 = ?.
```

ProDebug consists of two main components: a fault localizer, which identifies faulty clauses, and a repair module, which generates corrections for the identified bugs. A report is then generated and provided to the student based on this information.

The fault localizer supports three distinct methods: (1) Spectrum-based fault localization (SBFL), (2) mutation-based fault localization (MBFL), and (3) a method using large language models (LLMs). While concepts such as SBFL and MBFL are not new, our application to the Prolog context diverges significantly from previous approaches. Distinct from works that rely on mutation analysis to estimate coverage (Thompson and Sullivan 2020), ProDebug implements a novel SBFL engine that leverages low-level Prolog tracing Application Programming Interfaces (APIs). This allows us to capture exact execution spectra, including backtracking paths, which is critical for accurate localization.

Likewise, the repair module also supports multiple approaches: a systematic mutation-based search using program synthesis (Brancas *et al.* 2025), and a fine-tuned LLM. To the best of our knowledge, our mutation engine is the first application of constraint-based, syntax-guided mutation to Prolog, enabling the enumeration of non-redundant mutations. Once a candidate repair is obtained, the differing parts of the programs are abstracted with question marks, producing hints that highlight where students need to revise their code.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

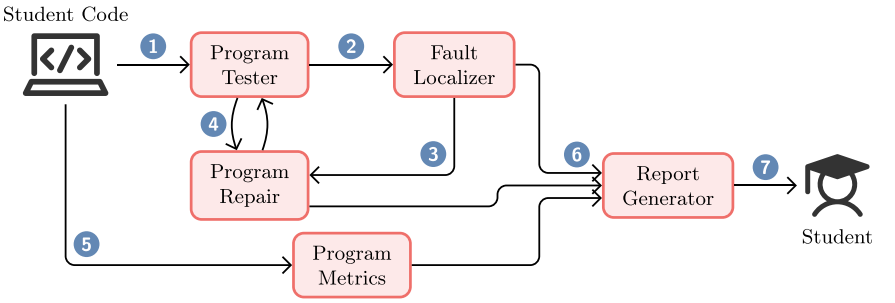

Fig. 1. System architecture overview.

This paper makes the following contributions:

- An automated fault localization approach for Prolog programs that integrates spectrum-based, mutation-based and LLM-based approaches;
- An automated repair procedure for Prolog that combines LLM-based patching with synthesis-driven mutation;
- The first fully automated tool for Prolog that offers both fault localization and program repair in an educational setting.[1]

2 System architecture

ProDebug is an integrated tool designed to help Prolog students understand and fix problems in their coding assignments. ProDebug was developed as a Prolog extension for GitSEED (Orvalho *et al.* 2024), an open-source automated assessment platform. When students submit their code to a GitLab repository where GitSEED is activated, the student's program is automatically evaluated, and a feedback report is generated and returned to the student.

Figure 1 shows the overall system architecture of ProDebug. When a student's program is submitted, first it is tested against a suite of unit tests (1). If the program is incorrect (i.e., fails some test), it is passed to the fault localizer (2) that attempts to find faults in the submission. Then, the fault report is sent to the program repair module (3), which uses that information to generate a patch to fix at least one of the program's bugs. This patch generation can take several attempts, and each attempt is tested against the test suite for correctness (4). Complementary to the main fault localization and repair modules, ProDebug also collects additional program-level metrics and information (5). These include code complexity features such as the *average clause length* or the *number of clauses per predicate*. The information from the fault localizer, repair module, and evaluation metrics is then collected into a report (6). This report can be customized by the faculty depending on the overall course design and the exercise's characteristics. Finally, the report is returned to the student (7) through GitSEED.

[1] Available at https://doi.org/10.5281/zenodo.18514417.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

```
1 parent(alice, bob).
2 parent(bob, carol).
3 grandparent(X, Z) :- parent(X, Y),
4                      parent(Y, Z).
```

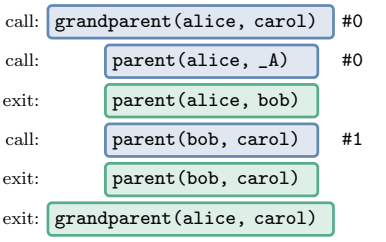

Fig. 2. On the left of this figure we show a Prolog program, while on the right we show an execution trace for the query `?- grandparent(alice, carol).`.

3 Fault localization

One of the main features of ProDebug is the automatic identification of buggy Prolog code. We accomplish this through three different fault localization methods: Spectrumbased Fault Localization (SBFL), Mutation-based Fault Localization (MBFL), and Large Language Model Fault Localization (LLMFL). Next, we describe each method in detail.

3.1 Spectrum-based fault localization

SBFL is a widely known approach to finding faults in imperative programming languages (Wong *et al.* 2016). Spectrum-based tools collect program execution traces (called *program spectra*) from passing and failing test cases. By comparing which program elements (e.g., statements) are executed during successful runs versus failed runs, SBFL assigns a suspiciousness score to each program element.

While spectrum-based methods are not directly applicable to declarative programming languages, due to the lack of an explicit control flow, we can exploit the mixed execution model of Prolog. Although Prolog programs have a declarative logical meaning, they are executed via a procedural mechanism based on unification, search, and backtracking, which can produce an observable trace.

Some Prolog interpreters, such as SWI-Prolog, have native support for extracting execution traces. These traces usually contain four types of information: `call` when a goal is started, `exit` when a goal is proved, `redo` when backtracking happens, and `fail` when a goal fails. In Figure 2, we show an example of a Prolog program, the instrumented version of that program, and an execution trace for the query `?- grandparent(alice, carol).`. In ProDebug, a *program element* corresponds to a Prolog clause (i.e., a rule or fact). Suspiciousness scores are thus assigned at the clause level, and spectrum information is extracted directly from the `call` and `redo` events of the trace.

Spectrum-based methods collect information from the execution of several tests. For each program element, s, four pieces of information are collected: number of tests that passed and s was executed (e_p), number of tests that passed and s was not executed (n_p), number of tests that failed and s was executed (e_f), and number of tests that failed and s was not executed (n_f). The intuition behind the SBFL method is that program elements that were disproportionately executed more often in failed test cases are more likely to be buggy. Crucially, the suspiciousness formulas compare *relative* execution

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Original clauses:

```
duplicate([], []).
duplicate([H|T], L2) :- duplicate(T, L1), L2 = (H, H, L1).
```

Example mutants:

```
duplicate(_, []).
duplicate([], V0).
duplicate([], 1).
duplicate([H | T], L2) :- duplicate(T, L1), L2 is (H, H, L1).
duplicate([H | T], L2) :- duplicate(T, L1), L2 = (H, _, L1).
duplicate([H | T], L2) :- duplicate(T, L1), L2 = [H, H, L1].
```

Fig. 3. Example of a program composed of two clauses and possible mutants generated by ProDebug for each of those clauses.

frequencies: a clause executed equally often across passing and failing tests receives a low score regardless of the absolute counts. In our setting, this is further supported by the use of test suites that include both positive tests (queries that should succeed) and negative tests (queries that should fail), ensuring that correct clauses are exercised in passing tests and providing a meaningful baseline for comparison. Several formulas exist for turning these metrics into a suspiciousness score for each element. ProDebug implements some of the most commonly used ones (Chatterjee *et al.* 2023): Tarantula, Jaccard, Ochiai, Barinel, Kulczynski, Op2 and DStar.

3.2 Mutation-based fault localization

Mutation-based Fault Localization (MBFL) is a fault localization approach that leverages the principles of mutation testing (Moon *et al.* 2014). Mutation-based tools generate a set of *mutants* by applying small syntactic changes to the program under test. These mutants are then executed against both passing and failing test cases to observe how the introduced changes affect program behavior. By analyzing which mutants are "killed" (detected by a test, meaning their behavior differs from the original program) and which survive, MBFL computes a suspiciousness score for each clause. Clauses whose mutants are disproportionately killed by failing test cases are considered more likely to contain the fault (Moon *et al.* 2014).

A central point of MBFL is generating the program mutants. ProDebug accomplishes this through a logic-based mutation enumeration engine using Satisfiability Modulo Theories (SMT) (Barrett *et al.* 2021). This mutation enumeration engine encodes Prolog Prolog Abstract Syntax Trees (ASTs) as logic formulas and then relaxes parts of those formulas in order to enumerate mutations (a detailed technical description is provided in Appendix B). While our approach is based on previous techniques for Answer Set Programming (ASP) bug repair (Brancas *et al.* 2025), adapting this method to Prolog requires addressing several key differences. Unlike ASP, where the order of rules and literals is irrelevant, Prolog's execution is sensitive to the order of clauses and body terms. Furthermore, the two languages differ significantly in operator and negation semantics (Negation as Failure in Prolog *vs.* Stable Model semantics in ASP). ProDebug also accounts for Prolog-specific features such as support for higher-order predicates, which are not present in ASP. Figure 3 shows some mutants generated by ProDebug for the two clauses of the introduction example program. Like for SBFL, several suspiciousness formulas exist for MBFL. ProDebug implements the two most commonly used ones: MUSE (Moon *et al.* 2014) and Metallaxis (Papadakis and Traon 2015).

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

3.3 Large language model-based fault localization

The third fault localization method included in ProDebug is LLMFL. This approach takes advantage of LLMs trained on large amounts of data and especially code. Due to the specific deployment requirements of ProDebug (real-time usage in classroom with possibly few resources), we focus on fine-tuned open-access *small* LLMs (∼4 billion parameters) in opposition to using closed-access models or very large instruction-tuned models. Our prompt format, shown in Appendix A, includes a description of the program's desired functionality, a reference implementation provided by the faculty, and the faulty student submission. The output format consists of a sequence of buggy clauses identified in the student's code.

To fine-tune our fault localization models, we used Low-Rank Adaptation (LoRA) (Hu *et al*. 2022) and Group Relative Policy Optimization (GRPO) (Shao *et al.* 2024). LoRA is a technique that uses rank decomposition matrices to greatly decrease the number of trainable parameters of a model. This technique has two main advantages: it reduces GPU VRAM requirements during training and it serves as a model regularizer, reducing the risk of overfitting. GRPO is an online reinforcement learning method for training neural networks. In GRPO, the model generates multiple candidate completions for each training example. A reward is then assigned to each completion using a user-defined reward function. Instead of optimizing each completion in isolation, GRPO compares them against one another: the relative advantages between completions are used to guide updates to the model's policy, encouraging it to prefer higher-reward outputs. For more details on GRPO, we refer to the original paper by Shao et al. (2024).

4 Program repair

The second main component of ProDebug is the program repair module. Given the fault localization report, its task is to find a fix for the bugs in the student submission. This fix can then be reported directly to the student or turned into a hint, depending on the faculty's preference. In this section, we describe two program repair approaches: one based on enumerating program mutations (Subsection 4.1) and another leveraging LLMs (Subsection 4.2).

4.1 Mutation-based repair

ProDebug's mutation-based repair is based on the MBFL mutation engine introduced in Subsection 3.2. Figure 4 shows a concrete example of how ProDebug enumerates program mutations. First, the Prolog program (or program section) is parsed and transformed into an AST (shown on the left side of Figure 4). Next, the AST is used to create a logic formula that represents the program (shown on the right side of Figure 4). By relaxing some of the constraints in the formula and using an SMT solver, we can generate new versions of the program with small differences (i.e., mutations). For instance, relaxing the constraint $n_7 =$ `L1` would allow the solver to enumerate the terms `duplicate(T, L2)` or `duplicate(T, H)`. ProDebug enumerates candidate repairs in increasing order of complexity: first one mutation, then two mutations, and so on.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

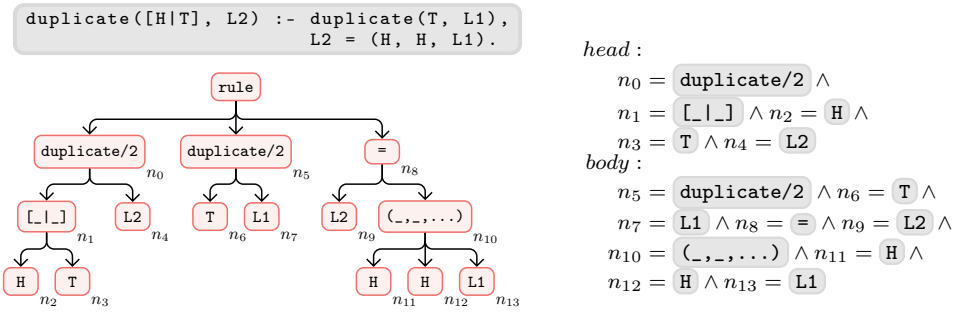

Fig. 4. A buggy Prolog rule, its AST representation, and the SMT encoding for that tree.

Using an SMT-based approach to generate mutations allows us to capture part of Prolog's semantics within the logical encoding. This enables ProDebug to constrain the search space by filtering out equivalent or semantically invalid mutations during enumeration. However, this expressiveness comes at the cost of increased computational overhead, as SMT-based enumeration typically incurs higher runtime compared to purely syntactic mutation generation techniques.

Before creating the SMT formula, the AST can be artificially expanded in order to support more diverse mutations. For example, by adding two empty nodes as children of n_9, we can enumerate the term `[T | L1] = (H, H, L1)`, which would be impossible otherwise. ProDebug automatically *completes* the AST by adding empty nodes so that all branches have the same depth and branching factor. Additionally, when the mutation engine is used for program repair, ProDebug enables some pruning constraints that help reduce the amount of equivalent and/or incorrect mutations. Appendix B provides a complete description of the encoding, including the relaxation mechanism, AST completion, and pruning constraints.

4.2 Large language model-based repair

Our LLM-based repair approach uses a prompt that contains a short description of the student's assignment, a reference implementation created by the faculty, the student's incorrect submission and the list of clauses reported by the fault localization module (see Appendix A). As in the LLM-based fault localization approach, we use small open-access LLMs to support classroom deployment.

For the repair fine-tuning, we used a combination of Supervised Fine-tuning (SFT) and GRPO. As introduced in Subsection 3.3, GRPO creates several completions for each prompt and learns based on the relative rewards attributed to each of those completions. However, if the initial model performs the desired task very poorly, it can happen for all the completions to receive very similar rewards (and even all be 0 in extreme cases). In such cases, the model may learn very slowly or fail to learn altogether. To address this problem, we performed a short run of classical SFT and then continue training with GRPO from there. Like for the LLM fault localization approach, we used LoRA to decrease VRAM requirements during training and to serve as a regularizer.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

5 Evaluation

We focus the evaluation of ProDebug on a large set of correct and incorrect Prolog programs collected in a bachelor's level logic programming class. Overall, we considered 1499 instances, each consisting of a buggy and a corrected program. Of these 1499 instances, 229 were collected from short practice exercises with an average of 2.9 clauses per exercise, while 1270 were collected from the longer final project assignment with an average of 33.3 clauses. Subsection 5.1 explains in further detail how these instances were obtained.

In this evaluation, we aim to answer the following research questions:

RQ1 How effective are the different fault localization methods?
RQ2 How do ProDebug's SBFL and MBFL compare with prior tools?
RQ3 How effective is the program repair?
RQ4 What is the impact of fine-tuning?

5.1 Methodology

5.1.1 Dataset

The original set of programs used to create the instances in this evaluation was collected in a bachelor's level logic programming class (Brancas *et al.* 2026). Each data point in the dataset corresponds to a Git commit containing two versions of a student's program: the previous version and the current one. Among all commits, 2495 were identified as *bugfix commits*, meaning the new version passed strictly more tests than the previous one. Of these 2495, we filtered out commits with syntax errors and those where the "fix" only involved adding a missing predicate that had not been implemented in the previous version. Such cases were excluded because they do not represent true bug fixes. Students often build their programs incrementally, implementing new predicates over time. As such, a missing predicate is typically a feature not yet implemented, rather than an actual error. After this filtering, we obtained 1499 instances, each consisting of a buggy and a corrected program.

To evaluate the performance of the different fault localization methods, we need to establish the ground truth, that is which clauses in a program are faulty. This is challenging because students might modify clauses that are not relevant to the fix or might not fix all bugs in the program. Therefore, we only include in the ground truth modified clauses that are relevant to the newly passed tests. We accomplish this by looking at the predicates invoked by these newly passed tests and computing their transitive closure over the program (i.e., all the predicates possibly called by the tests). The ground truth is then defined as the set of modified clauses that belong to these predicates.

5.1.2 Fault localization evaluation

The SBFL and MBFL approaches return a suspiciousness score for each clauses, where more suspicious clauses are assigned higher values. When several clauses are assigned the same suspiciousness, ProDebug uses the order they appear in the program to break ties, with elements that appear later being assigned higher ranks. The motivation for this is that students usually implement predicates sequentially and test as they go. As such, bugs are more likely to be present in later parts of the program.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

LLM fault localization methods directly return a ranking of suspicious lines. Lines not included in the ranking are placed below ranked lines in the inverse order they appear in the program, similarly to SBFL and MBFL disambiguation.

In the evaluation of the fault localization methods, we present three commonly used metrics (Maitama *et al.* 2020):

- MinRank: first position in the ranking with a faulty clause;
- Accuracy@k: percentage of faults in the program that appear in the first k positions of the ranking;
- Expense: percentage of the program a developer needs to go through (in the ranking order) until a fault is found. It is computed as the MinRank divided by the number of clauses.

5.1.3 Program repair evaluation

We consider a repair candidate correct if it flips at least one test to passing and does not flip any tests to failing. This follows the typical student use case where they only need a small hint in order to keep progress moving along. As such, when using mutation-based repair, the search terminates as soon as a correct repair candidate is found. However, inferencing several completions in the LLM repair method only takes a little longer than a single one due to latency, batching and caching. As such, in this method, we generate several completions and give preference to ones that fix the most bugs in the program. If there are several tied candidate repairs in the number of tests flipped, we choose the one that is closest to the original student program. The current implementation of ProDebug's repair only supports repairing bugs at the clause-level and not term-level.

5.1.4 LLM fine-tuning

In order to fine-tune the LLMs for fault localization and repair, we need a large number of diverse instances with accurate ground truths. Therefore, we developed a bug insertion tool that can create realistic-looking bugs in correct Prolog programs. This tool uses a combination of rule-based modifications and the mutation engine introduced in Subsection 3.2 to insert different bugs in correct programs. Based on the correct submissions from the dataset, we created 10,000 synthetic buggy instances to fine-tune the LLMs.

5.2 Implementation

ProDebug is implemented in Python and uses the SWI-Prolog dialect and interpreter through the Machine Query Interface. Parsing of Prolog programs is accomplished through a custom-built grammar. ProDebug uses a modified version of the Trinity framework (Martins *et al.* 2019) for the enumeration of mutations for fault localization, repair and creation of synthetic instances for fine-tuning. Results were obtained within a 10-minute time limit (wall clock time) and with 60 GB of RAM per instance, which was strictly enforced through `runsolver` (Roussel 2011) and `runhelper`.[2] We use a time limit of 10 minutes in this evaluation because we consider it the upper limit of time a

[2] https://pypi.org/project/runhelper/

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Table 1. *Different evaluation metrics for each fault localization method split among the practice exercises and the project submissions. Acc@k: percentage of faults ranked within the top k positions (higher is better); Expense: MinRank divided by the total number of clauses, i.e., the fraction of the program inspected before the first fault is found (lower is better)*

Instance type/method	MinRank	Acc@1	Acc@3	Acc@5	Acc@10	Expense	Timeouts
Practice exercises (n = 229)							
SBFL	**1.23**	83.57	**97.73**	–	–	48.29	4.4%
MBFL	1.42	70.95	96.19	–	–	53.71	5.7%
LLMFL	**1.23**	**84.75**	97.01	–	–	**47.53**	**0.0%**
ProFL SBFL	1.68	58.54	93.01	–	–	62.67	7.9%
ProFL MBFL	1.76	50.24	91.63	–	–	67.20	7.9%
Project (n = 1270)							
SBFL	3.33	**56.82**	**63.22**	**71.35**	82.74	**10.86**	9.1%
MBFL	3.76	45.08	57.58	70.05	**85.49**	16.05	59.9%
LLMFL	6.32	24.11	41.06	55.77	74.85	19.11	**0.0%**
ProFL SBFL	**3.28**	48.67	60.32	70.91	84.21	28.42	90.6%
ProFL MBFL	4.80	34.65	42.74	50.54	78.50	43.96	91.5%

student would accept to wait for automated feedback. Results were obtained on an Intel Xeon Silver 4210R processor. We employed the Qwen 3 family of models (Team 2025) for fault localization, and the Qwen 2.5 Coder family (Hui *et al.* 2024) for program repair. These model families were selected empirically based on their performance in preliminary experiments for each respective task. LLMs were fine-tuned on four Nvidia RTX A4000 GPUs, and inference was performed on a single RTX A4000. ProDebug's source code, data, and logs are publicly available.[3]

5.3 How effective are the different fault localization methods?

We evaluated the three fault localization methods proposed in Section Section 3 and present the results in Table 1 under the labels SBFL, MBFL and LLMFL. Of these methods, the one with the best overall performance when considering both types of instances is SBFL, achieving 83.6% Accuracy@1 in the Practice Exercises and 56.8% Accuracy@1 in the Project submissions. This technique also has a fairly low percentage of timeouts[4] (8.4%), meaning it can provide accurate answers in most instances. The second best performing technique is LLMFL, with a very similar performance in the practice exercises. However, this approach struggles with the longer context of Project submissions, only achieving an Accuracy@1 of 24.1%. The LLMFL approach also has the big advantage of not having any timeouts. Finally, the worst performing approach is MBFL, which is less accurate across the board. Furthermore, this approach has the drawback of having to generate and test large amounts of mutants, which scale linearly with the number of

[3] https://doi.org/10.5281/zenodo.18514417
[4] Timeouts in SBFL result from copying large traces from SWI-Prolog to Python through a socket and do not represent a fundamental limitation of the approach.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Table 2. *Fault localization performance across SBFL and MBFL formulas*

SBFL formula	Acc@1
Tarantula	55.75%
Ochiai	**63.70%**
Op2	60.96%
Barinel	55.75%
Jaccard	63.57%
DStar	50.31%
Kulczynski	50.57%
MBFL formula	**Acc@1**
Metallaxis	**52.87%**
MUSE	50.29%

clauses in the program. This is reflected in the much larger percentage of timeouts in the MBFL approach for the project (59.9%) *versus* for the practice exercises (5.7%).

Table 2 shows summarized results for the SBFL and MBFL approaches using the different spectrum formulas implemented in ProDebug. The best performing formula for SBFL is Ochiai, with Jaccard as a close second. For MBFL, the Metallaxis formula is slightly better than MUSE. Based on these results, ProDebug utilizes Ochiai and Metallaxis as its default formulas.

The most effective fault localization method is the Spectrum-based Fault Localization, followed by the LLM-based method and then the MBFL. The SBFL method correctly selects a buggy clause as the most suspicious in 84% of practice exercises and 57% of project submissions.

5.4 How do proDebug*'s SBFL and MBFL compare with prior tools?*

Table 1 also includes the results of the SBFL and MBFL approaches implemented by a previous Prolog fault localization tool, ProFL (Thompson and Sullivan 2020). Although both ProFL and ProDebug implement SBFL and MBFL methods, their implementations differ substantially. We highlight three key distinctions: (1) ProDebug communicates with Prolog via the Prolog Machine Query Interface, rather than launching a new Prolog process for each query, (2) ProDebug supports a broader range of program mutations through its SMT-based mutation engine, and (3) ProDebug measures program coverage directly using Prolog's tracing capabilities, whereas ProFL estimates clause coverage indirectly through program mutations. These differences address one of ProFL's major limitations: its need to enumerate mutations even for the SBFL approach, combined with launching a fresh process for each mutant, resulting in substantial execution overhead. Consequently, ProFL exhibits a high rate of timeouts across both approaches – particularly in the project submissions, where more than 90% of instances exceed the 10-minute timeout used in our evaluation.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Table 3. *Evaluation results for* ProDebug*'s program repair approaches*

	Practice exercises (n = 229)	Project (n = 1270)
Simulated perfect fault localization		
Mutation	21.0%	1.7%
LLM w/ 100 completions (default)	81.7%	34.3%
LLM w/ 30 completions	79.0%	26.1%
SBFL fault localization		
Mutation	20.2%	0.5%
LLM w/ 100 completions (default)	73.8%	30.9%
LLM w/ 30 completions	69.0%	15.0%

For the instances that finish within the time limit, ProDebug's SBFL approach outperforms ProFL's SBFL on the practice exercises (83.6% vs. 58.5% Acc@1), while both tools show comparable performance on the project submissions (56.8% vs. 48.7% Acc@1, with ProFL achieving a slightly better MinRank of 3.28 vs. 3.33). However, ProFL's MBFL approach performs consistently worse than ProDebug's MBFL across all instance types and metrics, likely due to its more limited set of generated mutants. Notably, ProFL's MBFL achieves only 50.2% Acc@1 on practice exercises compared to ProDebug's 70.9%, and 34.7% on project submissions compared to ProDebug's 45.1%.

Overall, ProDebug represents a significant advancement in Prolog fault localization, improving both the accuracy of fault identification and the efficiency of the analysis compared to previous tools.

5.5 How effective is the program repair?

To evaluate the effectiveness of our program repair methods, we tested them under two conditions: using a simulated *perfect* fault localization and using the best-performing fault localization technique identified in Subsection 5.3. This design allows us to assess both the idealized repair performance, as well as to quantify the extent to which repair success is constrained by inaccuracies in fault localization. Table 3 summarizes the outcomes for the two repair strategies, mutation-based and LLM-based, across these contexts.

Under simulated perfect fault localization, the mutation-based approach has a repair rate of 21.0% on practice exercises and 1.7% on project submissions. Although relatively low, these results are consistent with prior work on mutation-based repair for short declarative programs which achieved a 19% repair rate (Brancas *et al.* 2025). In contrast, the LLM-based repair method performs substantially better, reaching 81.7% for practice exercises and 34.3% for project submissions. As described in Subsubsection 5.1.4, the LLM repair method samples multiple completions and selects the best candidate among them. Depending on deployment constraints, the number of completions can be reduced to improve runtime at a modest cost to repair accuracy. For example, reducing from 100 to 30 completions decreases average repair time from 106 s to 77 s, while lowering

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Table 4. *Summarized results for fine-tuned and non fine-tuned Large Language Models. The models used for the fault localization were Qwen 3 with sizes 4B, 8B and 14B, while the models used for repair were Qwen 2.5 Coder with sizes 3B, 7B and 14B*

	Fault localization Acc@1	Repair rate
Fine-tuned LLM 4B/3B	33.4%	34.2%
Pre-trained LLM 4B/3B	14.2%	14.1%
Pre-trained LLM 8B/7B	6.5%	21.5%
Pre-trained LLM 14B	6.6%	37.9%

repair success by only 2.7 percentage points on exercises and 8.2 percentage points on projects.

When using the SBFL, repair performance decreases slightly. Mutation-based repair drops from 21.0% to 20.2% on exercises and from 1.7% to 0.5% on projects. The LLM-based method also declines, from 81.7% to 73.8% on exercises and from 34.3% to 30.9% on projects. The reduction is more pronounced for the configuration using only 30 completions, especially on the project dataset, suggesting that diversity in completions helps compensate for fault localization errors.

Overall, when using the SBFL localization, the LLM-based repair achieves the best performance, repairing 73.8% of practice exercises and 30.9% of project submissions. These results compare favorably with the state of the art in declarative program repair and suggest that advances in fault localization could further improve repair success.

5.6 What is the impact of fine-tuning?

In this work, we focused our LLM-based approaches on relatively small models with around 4 billion parameters. To evaluate the limitations of these smaller models and assess the impact of fine-tuning, we compared our fine-tuned 4B/3B models with larger non–fine-tuned models from the same families. Table 4 summarizes these results. As shown in Table 4, fine-tuned models consistently outperform their non–fine-tuned counterparts of the same size in both fault localization and repair. Moreover, the fine-tuned 4B/3B models achieve results comparable to or even surpassing those of much larger (8B–14B) pre-trained models.

Interestingly, the non–fine-tuned Qwen 3 4B model performs substantially better than its 8B and 14B counterparts. According to the Qwen developers, the 4B version underwent a longer and more extensive training process than the other two variants, which likely explains its stronger baseline performance.

Fine-tuning has a significant positive impact on model performance. Our fine-tuned models (4B for fault localization and 3B for repair) achieve results that match or exceed those of much larger 14B pre-trained models within their respective families.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

6 Related work

Debugging tools for Prolog. Prolog's declarative semantics often confuse learners accustomed to imperative control flow. To address this, researchers have developed environments that prioritize reasoning about program logic over procedural execution. GUPU (Neumerkel 1996) provides a sandboxed environment using program slicing and test-based diagnostics to help students identify logical inconsistencies. Subsequent systems have focused on automated feedback: PRAM (Mansouri *et al.* 1998) provides immediate correctness feedback via test suites, while INCOM (Le and Pinkwart 2011) uses weighted constraints to model student solutions and offer ranked hints based on common misconceptions.

Recent research has adapted imperative fault localization techniques to logic programming. ProFL (Thompson and Sullivan 2020) applies spectrum-based and mutation-based analyses to rank clauses by suspiciousness, while Terminyzer (Liang and Kifer 2013) specifically targets non-termination by analyzing recursive call chains. While effective at localization, these frameworks lack automated repair capabilities or machine learning-driven reasoning.

LLM-augmented debugging and repair. Recent research integrates LLMs with traditional debugging and repair methods. In declarative domains, FormHe (Brancas *et al.* 2025) combines logic-based fault localization, similarity metrics, program synthesis, and LLM reasoning to locate and repair faults in ASP programs. Both systems share the same SMT-based mutation approach and overall design philosophy of pairing logic-based analysis with LLM reasoning, though each system's mutation engine is adapted to its target language's syntax and semantics. However, they diverge in two key respects. For fault localization, FormHe exploits ASP's stable model semantics to identify minimal sets of rules responsible for a test failure, a technique with no counterpart in Prolog's procedural model; ProDebug instead introduces a novel SBFL engine based on Prolog's native tracing APIs. For LLM reasoning, FormHe fine-tunes models as classifiers, while ProDebug uses generative LLMs trained with GRPO, and is evaluated at a substantially larger scale (1499 submissions *vs.* 115). In a related direction, CodeHinter (Kurniawan *et al.* 2025) demonstrates that LLM-assisted feedback can improve debugging comprehension and learner confidence in introductory programming settings. These studies indicate that combining analytical debugging signals with LLM-generated insights yields more interpretable and actionable feedback.

ProDebug extends this trajectory to Prolog, unifying spectrum-based, mutation-based, and LLM-based reasoning to automatically localize and repair student faults. To our knowledge, ProDebug is the first tool to bridge traditional Prolog debugging with modern LLM repair methods.

7 Conclusion

This paper introduces ProDebug, an automated Prolog debugging system that combines spectrum-based, mutation-based, and LLM-based reasoning for fault localization and repair. Our evaluation on 1499 student submissions demonstrates that ProDebug effectively addresses the challenge of providing automated debugging in declarative programming education.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press

Key results show that Spectrum-based Fault Localization (SBFL) achieved the highest localization accuracy, identifying buggy clauses in 80% of practice exercises and 55% of project submissions. For program repair, fine-tuned LLMs significantly outperformed mutation-based methods, reaching a 74% success rate on practice exercises. These findings suggest that integrating logic-based analysis with generative reasoning provides a flexible debugging system suitable for classroom deployment. Future work will focus on multi-turn student interactions and leveraging term-level localization to further refine automated repairs.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068426100507

References

Barrett, C. W., Sebastiani, R., Seshia, S. A. and Tinelli, C. 2021. Satisfiability modulo theories. In *Handbook of Satisfiability - Second Edition 2021*, A. Biere, M. Heule, H. van Maaren and T. Walsh, Eds. Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 1267–1329.

Brancas, R., Manquinho, V. and Martins, R. 2025. Combining logic and large language models for assisted debugging and repair of ASP programs. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*, IEEE, 646–657.

Brancas, R., Orvalho, P., Carreira, C., Manquinho, V. and Martins, R. 2026. Can automated feedback turn students into happy prologians? In *Proceedings 42nd International Conference on Logic Programming, ICLP 2026, Lisbon, Portugal*, EPTCS.

Chatterjee, P., Campos, J., Abreu, R. and Roy, S. 2023. Augmenting automated spectrum based fault localization for multiple faults. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China* 2023, ijcai.org, 3140–3148.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L. and Chen, W. 2022. 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.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., Yang, A., Men, R., Huang, F., Ren, X., Ren, X., Zhou, J. and Lin, J. 2024. Qwen2.5-coder technical report. *CoRR, abs/2409.12186*.

Kurniawan, O., Chandra, E., Poskitt, C. M., Noller, Y., Choo, K. T. W. and Jégourel, C. 2025. Designing for novice debuggers: A pilot study on an AI-assisted debugging tool. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research, Koli Calling 2025, Koli, Finland, November 11-16, 2025*, J. Leinonen and R. Duran, Eds. ACM, 41:1–41:7.

Le, N.-T. and Pinkwart, N. 2011. Incom: A web-based homework coaching system for logic programming. In *Conference on Cognition and Exploratory Learning in Digital Age 2011*, 43–50. Citeseer.

Liang, S. and Kifer, M. 2013. Terminyzer: An automatic non-termination analyzer for large logic programs. In *Practical Aspects of Declarative Languages - 15th International Symposium, PADL 2013, Rome, Italy, January 21-22, 2013. Proceedings 2013*, K. Sagonas, Ed. Vol. 7752 of Lecture Notes in Computer Science, Springer, 173–189.

Maitama, J. Z., Idris, N. and Zakari, A. 2020. A systematic mapping study of the empirical explicit aspect extractions in sentiment analysis. *IEEE Access* 8, 113878–113899.

Mansouri, F. Z., Gibbon, C. A. and Higgins, C. A. 1998. PRAM: Prolog automatic marker. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 1998, Dublin City University, Ireland, 18-21 August 1998*, G. Davies and M. Ó'héigeartaigh, Eds. ACM, 166–170.

Martins, R., Chen, J., Chen, Y., Feng, Y. and Dillig, I. 2019. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment* 12, 12, 1914–1917.

Moon, S., Kim, Y., Kim, M. and Yoo, S. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA 2014*, IEEE Computer Society, 153–162.

Neumerkel, U. 1996. GUPU: A prolog course environment and its programming methodology (poster abstract). In *Logic Programming, Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, Bonn, Germany, September 2-6, 1996*, M. J. Maher, Ed. MIT Press, 549.

Orvalho, P., Janota, M. and Manquinho, V. 2024. GitSEED: A git-backed automated assessment tool for software engineering and programming education. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024, Virtual Event, NC, USA, December 5-8, 2024*, M. Dorodchi, M. Zhang and S. Cooper, Eds. ACM.

Papadakis, M. and Traon, Y. L. 2015. Metallaxis-FL: Mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 605–628.

Roussel, O. 2011. Controlling a solver execution with the runsolver tool: System description. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4, 139–144.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y. and Guo, D. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. CoRR, abs/2402.03300.

Team, Q. 2025. Qwen3 technical report. CoRR, abs/2505.09388.

Thompson, G. and Sullivan, A. K. 2020. ProFL: A fault localization framework for prolog. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 561–564.

Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8, 707–740.

https://doi.org/10.1017/S1471068426100507 Published online by Cambridge University Press