What Bugs Do Prolog Students Write? An Empirical Taxonomy and Data-Driven Mutation Framework

Ricardo Brancas
INESC-ID / IST,
Universidade de Lisboa,
Portugal

Pedro Orvalho
Artificial Intelligence Research Institute,
Consejo Superior de Investigaciones Científicas,
Barcelona, Catalonia, Spain

Carolina Carreira
Carnegie Mellon University,
Pittsburgh, USA
INESC-ID / IST, Universidade de Lisboa, Portugal

Vasco Manquinho
INESC-ID / IST,
Universidade de Lisboa,
Portugal

Ruben Martins
Carnegie Mellon University,
Pittsburgh, USA

Automated feedback tools for logic programming education depend on realistic bug datasets that reflect the mistakes students actually make. However, existing mutation testing frameworks for Prolog treat all mutations as equally likely, producing synthetic faults that diverge from classroom reality. We present an empirical study of 7,201 Prolog submissions from 265 undergraduate students, from which we derive a fine-grained taxonomy of student bugs through manual classification of 200 bug-fixing submissions. Guided by this taxonomy, we develop LOGMORPH, a data-driven mutation tool whose 17 operators are weighted according to the observed error distribution. LOGMORPH enumerates valid mutation sites on the abstract syntax tree, samples operators proportionally, injects faults, delegating to an SMT-based synthesizer when new code fragments are needed, and validates each mutant against a reference test suite. An evaluation of 16,000 generated mutants shows that the synthetic error distribution closely matches the student distribution, with most bug categories agreeing to within two percentage points. We identify cut-related mutations and synthesizer-generated code as the main sources of residual divergence, and outline how combining the SMT back-end with a language model fine-tuned on student code can further improve realism.

1 Introduction

Teaching logic programming presents a distinctive pedagogical challenge. Unlike imperative or object-oriented paradigms, where students can reason about programs by mentally simulating step-by-step execution, Prolog demands a fundamentally different mode of thinking: declarative specification, unification, and recursive descent through logical clauses. This shift is a well-documented source of difficulty [12, 31, 29, 8], and the resulting errors are often different from those in mainstream languages [14]. Understanding their nature, frequency, and structure is essential for building effective educational tools, yet the empirical landscape of student bugs in logic programming remains underexplored.

Recently, automated feedback tools for programming education have made significant strides [23, 27, 16, 17, 24], offering students immediate guidance on their submissions. However, the effectiveness of such tools depends critically on how well they anticipate the kinds of mistakes students actually make. A feedback system trained on uniformly distributed synthetic bugs may perform well in aggregate but miss the long-tailed, idiosyncratic errors that dominate real classrooms [3, 9], a mismatch that is a core obstacle for robust tutoring systems.

Mutation testing [11, 18], the systematic injection of small syntactic faults into correct programs, is a natural approach to generating buggy variants for training and evaluation purposes. However, traditional

To appear in EPTCS.

mutation frameworks are designed with software testing in mind: they aim for exhaustive coverage of a predefined set of operators, treating all mutations as equally likely. This assumption is poorly suited to educational contexts, where certain mistake patterns (such as forgetting a base case clause) vastly outnumber others (such as misplacing a cut operator). A more faithful simulation of student behavior requires grounding the mutation process in empirical data about which mistakes students actually make.

In this paper, we address this gap through two complementary contributions. First, we conduct a detailed empirical study of student submissions from a 9-week bachelor's-level logic programming course. We analyze 7201 Prolog code submissions from 265 students across three types of assignments (a role-playing exercise, recitation exercises, and a graded final project) and produce a fine-grained taxonomy of bug types observed in 200 real student programs. Our analysis reveals, among other findings, that incomplete implementations account for the largest share of errors (37.5%), followed by wrong argument usage (20.5%) and rule goal problems (13.0%), with other categories such as operator errors, incorrect predicate names, and cut-related issues occurring at lower frequencies.

Second, we leverage this empirical distribution to build LOGMORPH, a data-driven mutation tool for Prolog. Unlike conventional mutation frameworks, LOGMORPH samples mutation operators according to the observed frequency of each bug type in the student dataset, producing synthetic buggy programs whose error profiles match those of real learners. LOGMORPH operates over a custom Abstract Syntax Tree representation of Prolog programs and implements a four-stage pipeline (enumeration, sampling, injection, and validation) that ensures every generated mutant is both syntactically valid and semantically distinct from the original. For mutations that require generating new code fragments (such as inserting a spurious goal or replacing an argument), LOGMORPH delegates to an SMT-based program synthesizer that produces structurally valid Prolog terms respecting the context of the mutation site.

The contributions of this paper are as follows:

1. **An annotated dataset of student Prolog submissions.** We release a publicly available dataset[1] of 7201 submissions from 265 students, annotated with correctness labels and progression categories (bug fixed, bug introduced, mixed, no change). This dataset provides a foundation for future research on student behavior in logic programming courses.

2. **A taxonomy of student bugs in Prolog.** Through manual classification of 200 randomly selected bug-fixing submissions, we identify and categorize the most common error patterns, including subtypes within each major category. This taxonomy offers both quantitative frequency data and concrete examples drawn from real student code.

3. **A data-driven mutation tool (LOGMORPH).** We present a mutation framework that uses the empirically derived bug distribution to generate realistic buggy Prolog programs. LOGMORPH's weighted sampling mechanism, combined with a validation loop that rejects trivial or syntactically invalid mutants, produces a synthetic dataset that is representative of actual classroom errors.

4. **An empirical evaluation.** We compare the distribution of bug categories in a synthetic dataset[1] of 16,000 mutants against the empirical distribution observed in student submissions, showing close agreement for the majority of categories, and perform a qualitative analysis of the generated mutations, identifying both their strengths and current limitations.

[1] `https://figshare.com/s/fca6cb79db0790e85deb`

2 Related Work

Student error taxonomies. Empirical classification of novice programming errors has a long history. Pea [25] identified language-independent conceptual bugs, while Spohrer and Soloway [28] showed that plan-composition errors dominate in Pascal. For Java, Altadmri and Brown [3] analyzed over 37 million compilations from the Blackbox dataset, and Brown and Altadmri [9] showed that educator beliefs about common errors diverge from empirical evidence. Albrecht and Grabowski [2] manually classified 12,371 C submissions, finding that many errors stem from carelessness rather than misconceptions. In logic programming, the challenges of unification and backtracking are well documented [31, 29], and the special issue of *Instructional Science* [8] advocated structured programming techniques to mitigate ad hoc coding errors. Nevertheless, quantitative bug data remain scarce: Fung et al. [14] proposed a qualitative taxonomy of Prolog misconceptions, focusing on execution-model misunderstandings rather than code-level patterns. Our work addresses this gap with a quantitative bug taxonomy grounded in 7,201 real student submissions.

Mutation testing. Mutation testing was introduced by DeMillo et al. [11] and Hamlet [18], who proposed systematically injecting small faults to evaluate test suites. More recently, Andrews et al. [4] and Just et al. [21] found significant correlations between mutant and real-fault detection, while Gopinath et al. [15] showed that fault distributions differ across languages, motivating language-specific operators. For Prolog, Toaldo and Vergilio [30] defined the first mutation operators based on common logic programming mistakes, and Efremidis et al. [13] built a mutation testing framework for SWI-Prolog and SICStus. Neither Prolog-specific work weighted operators by empirical error frequencies. Our tool derives operator weights from observed classroom data, following the data-driven philosophy of Beller et al. [5], whose Mutation Monkey at Facebook learns patterns from real bug corpora.

Automated feedback, repair, and debugging. Automated feedback for programming education encompasses synthesis-based repair [27, 16, 32], semantics-preserving refactoring [20], verified repair via MaxSMT [1], and neural approaches [17, 6]. All of these target imperative languages. For logic programming, Hong [19] proposed grammar-based error analysis for a Prolog tutor, and Brancas et al. [7] combined logic-based techniques with large language models for Answer Set Programming. On the debugging side, algorithmic debugging of logic programs was pioneered by Shapiro [26] and surveyed by Caballero et al. [10]. While such tools diagnose bugs in existing programs, our work instead generates realistic buggy programs, a complementary capability for training and evaluating feedback and debugging methods.

3 Methodology and Dataset

We evaluated students' behavior during a 9-week-long bachelor's level logic programming course. During this course, students learned classical logic reasoning, as well as logic programming through Prolog. Overall, students faced three different types of Prolog exercises: a 5-part role-playing exercise to familiarize them with the language, 22 optional recitation exercises, and one mandatory, graded final project. The mandatory project was worth 50% of the final grade, while the remaining 50% was from other theoretical assessments. Next, we describe the Prolog exercises in more detail.

- **Role-playing Exercise**: Optional, not graded, online only; 5 puzzles over 5 days; a series of simple logic puzzles where students help a detective catch a group of criminals using logic programming.

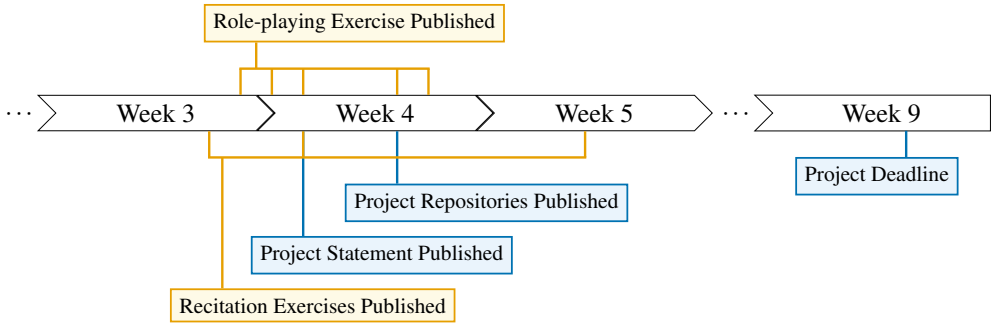


Figure 1: Course exercises timeline.

- **Recitation Exercises**: Optional, not graded, partially solved in-class using pen-and-paper; 22 exercises; 4 Prolog exercise sheets grouped into the topics of: lists, arithmetic, negation, and higher-order functions.
- **Project**: Mandatory, graded, worth 50% of the final grade; 5 weeks duration; the goal of this year's project was to create a solver for the logic puzzle *star battle*.

In Figure 1, we show the timeline of the different exercises solved by the students. During the first few weeks, the students learned logic fundamentals. Then, in week 3, students started learning Prolog, with optional exercise sets released during weeks 3, 4, and 5. Meanwhile, the project statement was released at the beginning of week 4 and lasted until the end of week 9.

3.1 Dataset Description

Of the 312 students who made submissions, 265 gave permission for their code to be used for research purposes. Over the course, those 265 students made a total of 7201 code submissions to the different repositories. We have created a publicly available annotated dataset containing these submissions.[2]

Table 1 shows the total number of correct and incorrect submissions per assignment type, along with the average number of clauses in the students' submissions for each one (per-exercise breakdown is provided in Table 4). Overall, 1680 submissions passed all tests for the assignment, and 5521 failed at least one test. By far, the assignment with the most number of submissions is the final project, which is easily explained by it being the only mandatory and graded assignment. The ratio of correct to incorrect submissions also varies greatly, with the optional assignments having much fewer incorrect submissions than the project. There are two plausible explanations for this: (1) the optional assignments are easier, and thus take fewer tries to get right, and (2) only more engaged students solved the optional exercises.

3.2 Submission Analysis

Figure 2 shows the number of submissions for each day of the course and reveals interesting patterns in students' behavior. Particularly, most students only start solving the optional exercises on the platform once the project submissions are opened. It is also possible to see the number of submissions escalate over the three days leading up to the project deadline.

[2] `https://figshare.com/s/fca6cb79db0790e85deb`

Assignment	**Correct**	**Incorrect**	**Total**	**Avg. Clauses**
Role-playing Exercise	271	239	510	3.51
Recitation Exercises	969	442	1411	2.50
Project	440	4840	5280	34.20
Total	1680	5521	7201	25.81

Table 1: Number of correct and incorrect submissions per assignment type. See Table 4 in Appendix A for the per-exercise breakdown.

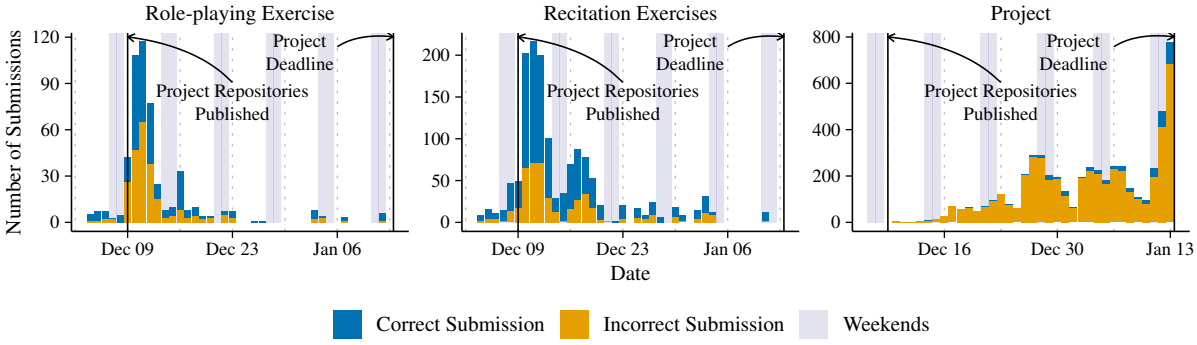


Figure 2: Number of submissions over time for each type of assignment.

We analyzed the 7201 submissions and classified them along two dimensions: (1) whether the submission is correct or incorrect, and (2) whether the submission is an improvement or a regression compared with the previous version of the code. For this second category, we consider 5 different labels:

- **First Submission**: first submission of a student for a particular assignment;
- **Bug Fixed**: the submission passes strictly more tests than the previous version;
- **Bug Introduced**: the submission passes strictly fewer tests than the previous version;
- **Mixed**: the submission both fails tests that previously passed and passes tests that previously failed;

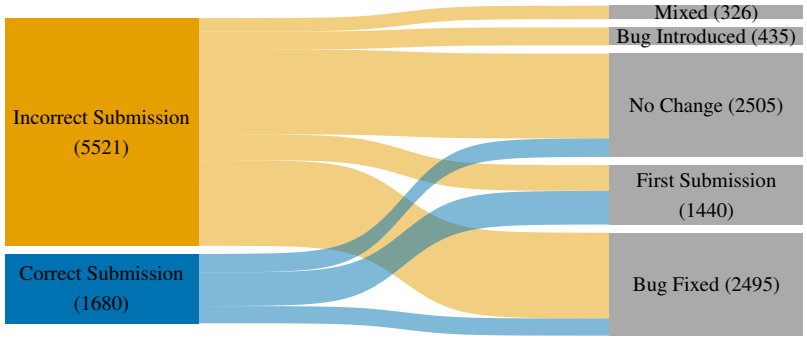


Figure 3: Classification of the 7201 student submissions.

- **No Change**: the submission passes exactly the same tests as the previous version.

Figure 3 shows the number of submissions in each category. We observe that most first submissions are correct, but, as evidenced by Table 1, this is largely due to the optional exercises. We note that the number of regressions is fairly small (Mixed, Bug Introduced). Lastly, there are a small number of submissions that are purely code quality improvements/refactorings of submissions that were already correct.

3.3 Bug Analysis

To understand the types and frequency of bugs, we analyzed submissions under the Bug Fixed label. Of those 2495 submissions, we isolated the ones in which only one predicate was modified – this subset represents "ideal" unit commits in which students fixed a specific incorrect behavior in their program. This selection resulted in 767 submissions, of which we manually classified 200 chosen randomly.

Table 2 presents the identified bug types and subtypes found in the 200 selected programs, along with their frequencies and a representative example for each type. Note that a single program may contain multiple bugs, and thus the total number of reported bug types exceeds 200. The most commonly identified reason for failing submissions was "Incomplete" programs, which can happen for two reasons: (1) students implemented their programs incrementally and tested components as they progressed (most often represented by the "Missing Predicate" subtype), and (2) students forgot to implement some functionality (most often represented by the "Missing Clause" subtype). While the first case does not necessarily represent a true bug, it still gives us a relevant insight: students seem to prefer having many tests to evaluate each functionality in isolation.

Other bug types were more evenly distributed, with no specific bug occurring at an overwhelmingly high frequency. Of particular note is the "Domain Logic Problem" bug type, which reflects a misunderstanding of the intended functionality for a given predicate. Additionally, 13 of the 26 programs categorized as "Other" contain at least one predicate that is fundamentally incorrect and requires complete rewriting.

4 Replicating Student Bugs

To simulate student behavior and generate a large dataset of buggy Prolog programs, we developed LOGMORPH, a mutation tool. Unlike traditional mutation tools that aim to be exhaustive or statistically uniform, LOGMORPH is *data-driven*: it uses the frequency distribution of bug types identified in section 3 to generate mutations representative of actual student errors. LOGMORPH is implemented in Python and SWI-Prolog using a custom Prolog parser built for this work.

4.1 Architecture

The architecture of LOGMORPH is depicted in Figure 4. The pipeline transforms a correct reference implementation into a buggy variant through four stages:

Enumeration. The process begins with a *Correct Program* as input. The *Bug Position Collector* parses the source code into an Abstract Syntax Tree (AST) (1) using our custom parser and traverses the tree to identify all syntactically valid locations where mutation operators can be applied. For instance, the collector identifies every arithmetic operation as a candidate for operator mutation, and every predicate call as a candidate for argument swapping. This step yields a list of *Bug Type/Position Pairs* (2).

Bug Type	**Count**	**Frequency**
Incomplete	**75**	**37.5%**
Subtypes: Missing Predicate	59	29.5%
Missing Clause (of existing predicate)	16	8.0%
There are missing predicates/clauses. Example:		
`+mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 is E1 * N, mult(L1, N, L2).`		
Wrong Argument	**41**	**20.5%**
Subtypes: Argument Order Swap	7	3.5%
Missing Argument	6	3.0%
Extra Argument	2	1.0%
Other	26	13.0%
Issue in the arguments of a predicate call. Example:		
`mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 is E1 * N, mult(L1, N, ~~L1~~L2).`		
Rule Goal Problems	**26**	**13.0%**
Subtypes: Extra Goal	13	6.5%
Missing Goal	10	5.0%
Goal Order Swap	3	1.5%
Problems with the goals in the body of a rule. Example:		
`mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 is E1 * N, mult(L1, N, L2).`		
Operator Error	**23**	**11.5%**
Subtypes: Wrong Operator	16	8.0%
List Terminators Issue	4	2.0%
Missing Negation	3	1.5%
Incorrect operator usage. Example:		
`mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 ~~=~~is E1 * N, mult(L1, N, L2).`		
Wrong Variable/Constant	**21**	**10.5%**
Subtypes: Wrong Variable Name	19	9.5%
Wrong Constant	2	1.0%
Problems with the names of variables or constants. Example:		
`mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 is E1 * ~~M~~N, mult(L1, N, L2).`		
Wrong Predicate Name	**19**	**9.5%**
Incorrect predicate name in call. Example:		
`mult([], _, []).`		
`mult([E1|L1], N, [E2|L2]) :- E2 is E1 * N, mult~~iply~~(L1, N, L2).`		
Domain Logic Problem	**16**	**8.0%**
Different problems on the implementation of domain logic.		
Cut Problem	**14**	**7.0%**
Subtypes: Missing Cut	10	5.0%
Extra Cut	3	1.5%
Wrong Placement	1	0.5%
Problems with the cut operator (`!`). Example:		
`max(X, Y, X) :- X >= Y, !.max(_, Y, Y).`		
Other	**26**	**13.0%**

Table 2: Description and frequency of the different bug types manually identified in student submissions.

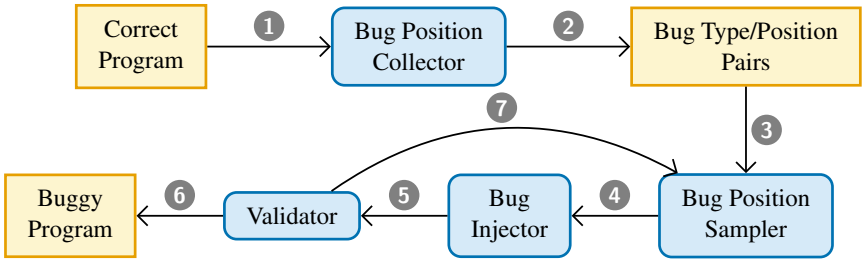


Figure 4: Architecture of LOGMORPH.

Sampling. The *Bug Position Sampler* receives the list of candidate mutations (3) and selects one according to the weighted probability distribution derived from our manual bug classification (see Table 2). This data-driven weighting ensures that the synthetic dataset mirrors classroom reality; for example, "Clause Deletion" mutations are generated significantly more often than cut mutations.

Injection. The *Bug Injector* takes the selected pair and rewrites the corresponding AST node to introduce the fault (4). For operators that require new code (e.g., goal or argument insertion), the injector delegates term generation to a program synthesizer. The synthesizer is SMT-based and built on the Trinity synthesis framework [22], following the same approach used by FORMHE [7] for Answer Set Programming. Given the type context of the target position, it generates a syntactically valid Prolog term that satisfies the structural constraints of the surrounding clause.

Validation. The mutated program is passed to the *Validator* (5), which checks that the injected bug produces a *non-trivial* and *observable* fault. Concretely, the validator runs the mutant against the reference test suite and rejects it if it passes all tests (i.e., the mutation is semantically neutral) or fails to load (i.e., the mutation is syntactically invalid). If the mutant is rejected, the validator signals the *Bug Position Sampler* to draw a new candidate (7), creating a rejection-sampling loop. Once a valid mutant is found, the pipeline emits the final *Buggy Program* (6).

4.2 Mutation Operators

We implemented a set of mutation operators that map directly to the bug categories identified in section 3, with sampling weights proportional to the observed frequencies in Table 2. One deliberate exception is the *Missing Predicate* subtype of "Incomplete" bugs: as discussed in section 3, these submissions typically reflect students testing partial implementations incrementally rather than genuine programming errors, and are therefore excluded. After removing the 59 submissions whose only bug was a missing predicate, 141 programs remain as the base for computing relative frequencies. The denominator in each weight is not always 141 because not all bug types are applicable to all programs; for instance, only 31 of the 141 programs use negation, so the *Missing Negation* frequency is computed over those 31 programs.

For easier comparison with the student data, fine-grained mutation operations are grouped into broader shared categories: "Domain Logic Problem", "Missing Predicate Call", "Missing Arithmetic Evaluation", "Goal Failure Suppression", and "Extraneous Code" are all mapped to the *Other* category, while "Missing List Terminators" and "Extra List Terminators" are merged into *List Terminators Issue*. Table 3 lists all implemented operators together with their weights and the shared category label.

Category	Operator	Description	Rel. Freq.	Weight
Incomplete	Clause Deletion	Remove a clause	16/141	11.35
Wrong Var. Name	Variable Repl.	Replace a variable in scope	19/141†	7.48
Wrong Constant	Constant Repl.	Replace a constant	2/99	2.02
Wrong Argument	Argument Mutation	Replace an argument	26/141‡	14.44
Missing Argument	Argument Removal	Remove an argument	6/141	4.26
Extra Argument	Argument Addition	Add an argument	2/141	1.42
Swap Argument	Argument Swap	Swap two arguments	7/136	5.15
Extra Goal	Goal Insertion	Insert a sub-goal	13/141	9.22
Rule Goal Swap	Goal Swap	Reorder sub-goals	3/134	2.24
Missing Goal	Goal Deletion	Remove a sub-goal	10/141	7.09
Wrong Operator	Operator Mutation	Change an operator	16/120	13.33
Missing Negation	Negation Removal	Remove \+	3/31	9.68
List Term. Issue	Missing List Term.	Remove list brackets	3/129	2.33
	Extra List Term.	Add list brackets	1/141	0.71
Missing Cut	Cut Removal	Remove a cut (!)	10/79	12.66
Extra Cut	Cut Addition	Add a cut (!)	3/141	2.13
Wrong Pred. Name	Predicate Rename	Replace a predicate name	19/141	13.48
Other	Missing Pred. Call	Remove a predicate call	1/141	0.71
	Missing Arith. Eval.	Remove is evaluation	1/89	1.12
	Goal Failure Suppr.	Suppress goal failure	5/27	18.52
	Extraneous Code	Insert extraneous code	1/141	0.71
	Code Injection/Repl.	Replace an AST node	18/141	12.76

† Bias: −0.06 ‡ Bias: −0.04. Weight includes bias.

Table 3: Mutation operators, descriptions, and sampling weights. The *Category* column shows the shared label used for comparison with student data in Figure 6. Operators marked with † and ‡ include a manual bias correction (see text).

Two weights include a manual correction to compensate for a systematic bias introduced by the validation step. Because the sampler and validator operate in a rejection-sampling loop, operators with a high validation pass rate are effectively over-sampled in the final output, while operators that are frequently rejected are under-sampled. Concretely, the *Variable Replacement* weight is reduced by 6 percentage points, and the *Argument Mutation* weight by 4 percentage points relative to their observed frequencies, since variable and argument replacements almost invariably produce detectable failures and therefore pass validation at a near-100% rate.

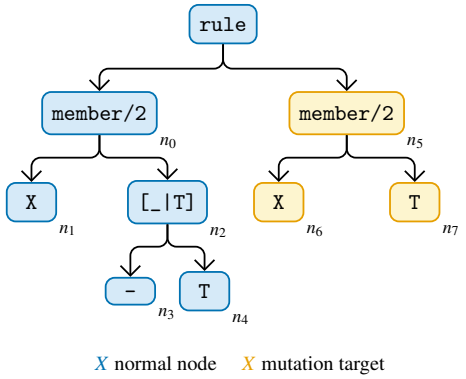
4.3 Worked Example

We trace the pipeline on a simple `member/2` predicate to make each stage concrete.

Input. The input is the following correct two-clause definition:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

Enumeration. The parser reads the source and builds an AST for each clause. Figure 5 shows the AST for clause 2. The root `rule` node has one child per element of the clause: the head (n_0) and each body goal (here only n_5). Each functor node carries its arguments as children, and list patterns such



Bug Type	Nodes
clause_deletion	root
argument_swap	n_0: args(n_1, n_2)
var_replacement	n_1
goal_deletion	n_5
predicate_rename	n_5
argument_swap	**n_5: args(n_6, n_7)**
var_replacement	n_6
var_replacement	n_7
⋮	

Figure 5: AST for clause 2 of `member/2` (left) and a subset of the bug type/position pairs (right). The bold row is the pair selected by the sampler; highlighted nodes (n_5–n_7) are the mutation target.

as `[_|T]` are represented as a two-child node (n_2) with an anonymous variable (n_3) and the tail variable (n_4). The *Bug Position Collector* traverses the AST and emits one pair for every applicable operator at every applicable node. A representative subset for `member/2` is listed in Figure 5.

Sampling. The sampler draws from the candidate list using the weighted distribution calibrated to our bug data. Suppose it selects the bold row in Figure 5: $\langle$`argument_swap`, n_5 : `args`$(n_6, n_7)\rangle$

Injection. The injector locates node n_5 in the AST and swaps its argument children n_6 and n_7, turning `member(X, T)` into `member(T, X)`, yielding:

```
member(X, [X|_]).
member(X, [_|T]) :- member(T, X).  % args swapped
```

Validation. The mutant is run against the reference test suite. `member(a,[a,b,c])` is still handled by the base clause, but the recursive call now diverges on `member(b,[a,b,c])`, so the mutant correctly fails tests and is accepted by the validator.

5 Analysis

We evaluate LogMorph along two dimensions: how well the synthetic bug distribution matches real student errors, and how realistic the generated mutations are in practice.

5.1 Distribution Comparison

We applied LogMorph to the reference solutions of the student exercises to generate a synthetic dataset of 16,000 buggy programs. Figure 6 compares the resulting distribution of bug categories against the empirical distribution observed in the student dataset.

Overall, the two distributions are in close agreement. For most categories, the difference between the population and synthetic percentages is less than 2 percentage points, confirming that the data-driven weighting mechanism successfully replicates the overall shape of real student error patterns. The three

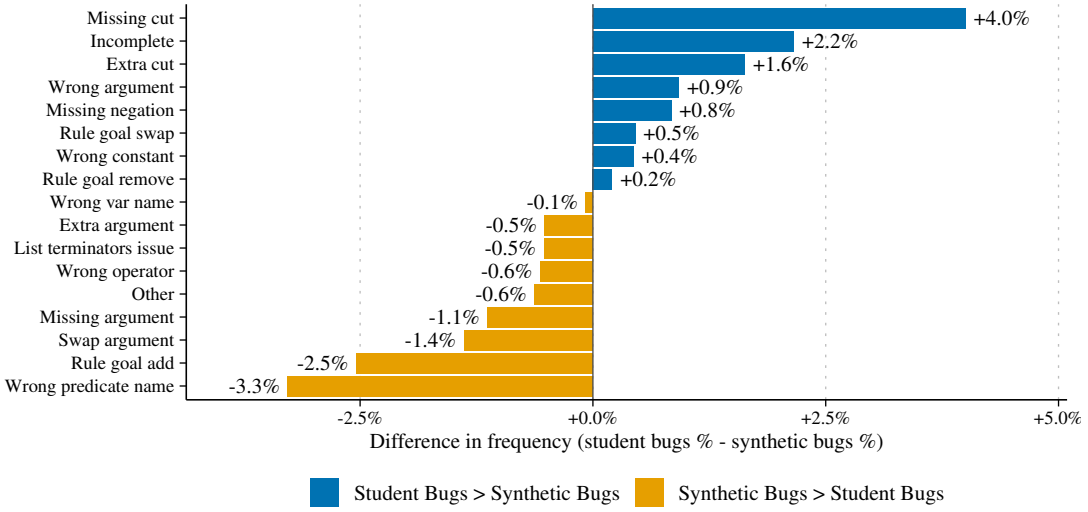


Figure 6: Distribution of bug categories in the student dataset (population) and in the 16,000 synthetically generated mutations (synthetic), as a percentage of each total.

most frequent categories, *Other* (20.79% vs. 21.42%), *Wrong argument* (15.35% vs. 14.43%), and *Wrong predicate name* (9.41% vs. 12.68%), account for roughly 45% of both the real and synthetic bugs.

The most pronounced discrepancy concerns cut-related bugs. In the student dataset, *Missing cut* accounts for 4.95% of bugs, and *Extra cut* for a further 1.98%, for a combined total of 6.93%. In the synthetic data, these categories shrink to 0.95% and 0.35%, respectively (combined: 1.30%). This under-representation is a direct consequence of the validation step: the rejection-sampling loop discards mutations that leave the test outcomes unchanged, and cut insertions or deletions are disproportionately rejected because many student predicates either do not use cuts or are not sensitive to them under the available tests.

In the opposite direction, *Wrong predicate name* is overrepresented in the synthetic data (12.68%) compared with the population (9.41%), a difference of 3.28 percentage points. *Rule goal add* shows a similar pattern (7.49% vs. 4.95%), likely because synthesized goals that do alter program semantics pass validation at a higher-than-average rate.

5.2 Qualitative Analysis and Limitations

We also examined individual mutations to assess their qualitative realism. For most operator types, the generated bugs are concise, local changes that closely resemble actual student errors. Consider the following three examples drawn from the synthetic dataset:

1. **Wrong operator.** In a list-multiplication predicate, the arithmetic evaluation operator `is` is replaced with the unification operator =:

```
multPorN([], _, []).
multPorN([H | T], N, [H1 | T1]) :- H1 ~~is~~= H * N, multPorN(T, N, T1).
```

2. **Wrong variable name.** In a list-duplication predicate, the tail variable `T` in the recursive call is replaced with the accumulator variable `X`:

```
duplicaElem([], []).
duplicaElem([H | T], L) :- duplicaElem(TX, X), L = [H, H | X].
```

3. **Missing recursive call.** The recursive goal is deleted from a clause body, turning the rule into a predicate that processes only the first element:

```
multPorN([], _, []).
multPorN([H | T], N, [W | WT]) :- W is H * N, multPorN(T, N, WT).
```

However, mutations that rely on the SMT-based synthesizer to generate new code fragments (specifically goal insertion, argument addition, and extraneous code injection) can produce visibly artificial results. Two representative examples illustrate this:

1. **Synthesized guard on a fact.** A goal insertion operator turns a base-case fact into a rule by adding the synthesized goal `[] < V0`:

```
eliminaNumeros([], []) :- [] < V0.    % guard added
```

Comparing an empty list with a fresh variable using < is syntactically valid but semantically nonsensical; no student would write such a guard.

2. **Extraneous synthesized clause.** An extraneous code operator injects an entire fabricated clause between two existing ones:

```
mult(N1, N2, N3) :- N3 is N1 * N2.
multPorN(_, _, _) :- [] * false = _,              % injected
                     multPorN(_, V0, _) =:= (\+[]).
multPorN(P1, N, P2) :- maplist(mult(N), P1, P2).
```

The injected clause exhibits typical traits of SMT synthesis: fresh variables (`V0`), Boolean constants (`false`), empty lists used as arithmetic operands, and operators applied to semantically incompatible arguments.

These artifacts arise because the Trinity-based synthesizer [22] explores the space of structurally valid terms without any model of idiomatic Prolog usage. Since Prolog is untyped, expressions such as `[] < V0` or `[] * false` are syntactically legal and accepted by the interpreter, so the validation step, which only checks whether the mutant produces different test outcomes, cannot reject them. The result is code that, while technically valid and behavior-altering, is immediately recognizable as artificial.

Replacing the SMT synthesizer with a large language model fine-tuned on student code would address this limitation: an LLM could generate insertions and replacements that respect both structural constraints and stylistic conventions, producing more realistic mutations for the operator types that currently require synthesis.

6 Conclusion

We have presented an empirical study of student errors in Prolog and LogMorph, a data-driven mutation tool that uses the resulting bug taxonomy to generate realistic buggy programs. Our evaluation shows that LogMorph's weighted sampling mechanism produces synthetic datasets whose distribution closely matches that of real student errors, with most bug categories differing by fewer than 2 percentage points.

The main open direction is to explore the use of large language models to complement the current SMT-based synthesizer, potentially enabling the generation of code fragments that better adhere to both

structural and stylistic conventions. We also plan to integrate LOGMORPH into automated feedback systems for logic programming and to test the approach with other logic programming languages and larger, multi-institutional student populations.

Acknowledgments

This work supported by Portuguese national funds through FCT, under project 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX). This work was also supported by grant PID2022-139835NB-C21 funded by MCIN/AEI/10.13039/501100011033 and by ERDF, EU; and by HORIZON-MSCA-2025-PF, project 101269051 — Sherlock4Py funded by REA, EU. This work was additionally supported by the Carnegie Mellon University Portugal Program and FCT under grants PRT/BD/152086/2021 (DOI: 10.54499/-PRT/BD/152086/2021) and PRT/BD/153739/2021 (DOI: 10.54499/PRT/BD/153739/2021). This work was also partially supported by the National Science Foundation (NSF) under Award CCF2427581 and DARPA under Agreement FA8750-24-9-1000.

References

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh & Abhik Roychoudhury (2022): *Verifix: Verified Repair of Programming Assignments*. *ACM Trans. Softw. Eng. Methodol.* 31(4), pp. 74:1–74:31, doi:10.1145/3510418.

[2] Ella Albrecht & Jens Grabowski (2020): *Sometimes It's Just Sloppiness - Studying Students' Programming Errors and Misconceptions*. In Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter & Alvaro E. Monge, editors: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020*, ACM, pp. 340–345, doi:10.1145/3328778.3366862.

[3] Amjad AlTadmri & Neil C. C. Brown (2015): *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data*. In Adrienne Decker, Kurt Eiselt, Carl Alphonce & Jodi L. Tims, editors: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education, SIGCSE 2015, Kansas City, MO, USA, March 4-7, 2015*, ACM, pp. 522–527, doi:10.1145/2676723.2677258.

[4] James H. Andrews, Lionel C. Briand & Yvan Labiche (2005): *Is mutation an appropriate tool for testing experiments?* In Gruia-Catalin Roman, William G. Griswold & Bashar Nuseibeh, editors: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, ACM, pp. 402–411, doi:10.1145/1062455.1062530.

[5] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra & Erik Meijer (2021): *What It Would Take to Use Mutation Testing in Industry - A Study at Facebook*. In: *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, IEEE, pp. 268–277, doi:10.1109/ICSE-SEIP52600.2021.00036.

[6] Sahil Bhatia, Pushmeet Kohli & Rishabh Singh (2018): *Neuro-symbolic program corrector for introductory programming assignments*. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik & Mark Harman, editors: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, ACM, pp. 60–70, doi:10.1145/3180155.3180219.

[7] Ricardo Brancas, Vasco Manquinho & Ruben Martins (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, pp. 646–657, doi:10.1109/ICST62969.2025.10988950.

[8] Paul Brna, Helen Pain & Benedict du Boulay (1990): *Teaching, Learning and Using Prolog: Understanding Prolog*. *Instructional Science* 19, pp. 247–256.

[9] Neil C. C. Brown & Amjad AlTadmri (2017): *Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs*. *ACM Trans. Comput. Educ.* 17(2), pp. 7:1–7:21, doi:10.1145/2994154.

[10] Rafael Caballero, Adrián Riesco & Josep Silva (2017): *A Survey of Algorithmic Debugging*. *ACM Comput. Surv.* 50(4), pp. 60:1–60:35, doi:10.1145/3106740.

[11] Richard A. DeMillo, Richard J. Lipton & Frederick G. Sayward (1978): *Hints on Test Data Selection: Help for the Practicing Programmer*. *Computer* 11(4), pp. 34–41, doi:10.1109/C-M.1978.218136.

[12] Benedict du Boulay (1986): *Some Difficulties of Learning to Program*. *Journal of Educational Computing Research* 2(1), pp. 57–73, doi:10.2190/3LFX-9RRF-67T8-UVK9.

[13] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings & Philipp Körner (2018): *Measuring Coverage of Prolog Programs Using Mutation Testing*. In Josep Silva, editor: *Functional and Constraint Logic Programming - 26th International Workshop, WFLP 2018, Frankfurt/Main, Germany, September 6, 2018, Revised Selected Papers*, Lecture Notes in Computer Science, Springer, pp. 39–55, doi:10.1007/978-3-030-16202-3_3.

[14] Patricia Fung, Mike Brayshaw, Benedict du Boulay & Mark Elsom-Cook (1990): *Towards a Taxonomy of Novices' Misconceptions of the Prolog Interpreter*. *Instructional Science* 19(4/5), pp. 311–336, doi:10.1007/BF00116443.

[15] Rahul Gopinath, Carlos Jensen & Alex Groce (2014): *Mutations: How Close are they to Real Faults?* In: *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, IEEE Computer Society, pp. 189–200, doi:10.1109/ISSRE.2014.40.

[16] Sumit Gulwani, Ivan Radicek & Florian Zuleger (2018): *Automated clustering and program repair for introductory programming assignments*. In Jeffrey S. Foster & Dan Grossman, editors: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, ACM, pp. 465–480, doi:10.1145/3192366.3192387.

[17] Rahul Gupta, Soham Pal, Aditya Kanade & Shirish K. Shevade (2017): *DeepFix: Fixing Common C Language Errors by Deep Learning*. In Satinder Singh & Shaul Markovitch, editors: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, AAAI Press, pp. 1345–1351, doi:10.1609/AAAI.V31I1.10742.

[18] Richard G. Hamlet (1977): *Testing Programs with the Aid of a Compiler*. *IEEE Trans. Software Eng.* 3(4), pp. 279–290, doi:10.1109/TSE.1977.231145.

[19] Jun Hong (2004): *Guided programming and automated error analysis in an intelligent Prolog tutor*. *Int. J. Hum. Comput. Stud.* 61(4), pp. 505–534, doi:10.1016/J.IJHCS.2004.02.001.

[20] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong & Abhik Roychoudhury (2019): *Re-Factoring Based Program Repair Applied to Programming Assignments*. In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, IEEE, pp. 388–398, doi:10.1109/ASE.2019.00044.

[21] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes & Gordon Fraser (2014): *Are mutants a valid substitute for real faults in software testing?* In Shing-Chi Cheung, Alessandro Orso & Margaret-Anne D. Storey, editors: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, ACM, pp. 654–665, doi:10.1145/2635868.2635929.

[22] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng & Isil Dillig (2019): *Trinity: An Extensible Synthesis Framework for Data Science*. *Proc. VLDB Endow.* 12(12), pp. 1914–1917, doi:10.14778/3352063.3352098.

[23] Pedro Orvalho, Mikolás Janota & Vasco Manquinho (2024): *GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education*. In Mohsen Dorodchi, Ming Zhang & Stephen Cooper, editors: *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024*, ACM, doi:10.1145/3649165.3690106.

[24] Pedro Orvalho, Mikolás Janota & Vasco Manquinho (2026): *MENTOR: Fixing introductory programming assignments with formula-based fault localization and LLM-driven program repair*. *J. Syst. Softw.* 234, p. 112690, doi:10.1016/J.JSS.2025.112690.

[25] Roy D. Pea (1986): *Language-Independent Conceptual "Bugs" in Novice Programming*. *Journal of Educational Computing Research* 2(1), pp. 25–36, doi:10.2190/689T-1R2A-X4W4-29J2.

[26] Ehud Y. Shapiro (1983): *Algorithmic Program Debugging*. MIT Press.

[27] Rishabh Singh, Sumit Gulwani & Armando Solar-Lezama (2013): *Automated feedback generation for introductory programming assignments*. In Hans-Juergen Boehm & Cormac Flanagan, editors: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, ACM, pp. 15–26, doi:10.1145/2491956.2462195.

[28] James C. Spohrer & Elliot Soloway (1986): *Novice Mistakes: Are the Folk Wisdoms Correct?* *Commun. ACM* 29(7), pp. 624–632, doi:10.1145/6138.6145.

[29] Jane Taylor (1990): *Analysing Novices Analysing Prolog: What Stories Do Novices Tell Themselves about Prolog?* *Instructional Science* 19(4/5), pp. 283–309, doi:10.1007/BF00116442.

[30] Jovelino Rosa Toaldo & Silvia Regina Vergilio (2006): *Applying Mutation Testing in Prolog Programs*. In: *VII Workshop de Testes e Tolerância a Falhas (WTF)*.

[31] Maarten W. Van Someren (1990): *What's Wrong? Understanding Beginners' Problems with Prolog*. *Instructional Science* 19(4/5), pp. 257–282, doi:10.1007/BF00116441.

[32] Ke Wang, Rishabh Singh & Zhendong Su (2018): *Search, align, and repair: data-driven feedback generation for introductory programming exercises*. In Jeffrey S. Foster & Dan Grossman, editors: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, ACM, pp. 481–495, doi:10.1145/3192366.3192384.

A Per-Exercise Submission Breakdown

Assignment	Correct	Incorrect	Total	Avg. Clauses
Role-playing Exercise	271	239	510	3.51
Day 1	88	58	146	3.00
Day 2	63	67	130	3.27
Day 3	40	68	108	5.05
Day 4	46	9	55	2.62
Day 5	34	37	71	3.26
Recitation Exercises	969	442	1411	2.50
Exercise A1	65	55	120	2.00
Exercise A2	79	44	123	2.67
Exercise A3	61	24	85	2.05
Exercise A4	49	15	64	2.00
Exercise A5	49	12	61	3.05
Exercise A6	50	21	71	2.00
Exercise A7	46	19	65	2.08
Exercise B1	57	42	99	3.09
Exercise B2	39	31	70	1.77
Exercise B3	42	16	58	3.55
Exercise B4	41	7	48	1.70
Exercise B5	45	15	60	2.86
Exercise C1	41	19	60	2.22
Exercise C2	36	6	42	2.98
Exercise C3	35	7	42	3.32
Exercise C4	37	17	54	3.11
Exercise D1	47	13	60	2.25
Exercise D2	35	10	45	2.56
Exercise D3	36	14	50	3.09
Exercise D4	33	23	56	3.02
Exercise D5	23	27	50	2.12
Exercise D6	23	5	28	2.00
Project	440	4840	5280	34.20
Total	1680	5521	7201	25.81

Table 4: Number of correct and incorrect submissions per exercise.