# ProFL: A Fault Localization Framework for Prolog

George Thompson
North Carolina A&T State University, USA
gmthompson1@aggies.ncat.edu

Allison K. Sullivan
The University of Texas at Arlington, USA
allison.sullivan@uta.edu

## ABSTRACT

Prolog is a declarative, first-order logic that has been used in a variety of domains to implement heavily rules-based systems. However, it is challenging to write a Prolog program correctly. Fortunately, the SWI-Prolog environment supports a unit testing framework, plunit, which enables developers to systematically check for correctness. However, knowing a program is faulty is just the first step. The developer then needs to fix the program which means the developer needs to determine what part of the program is faulty. ProFL is a fault localization tool that adapts imperative-based fault localization techniques to Prolog's declarative environment. ProFL takes as input a faulty Prolog program and a plunit test suite. Then, ProFL performs fault localization and returns a list of suspicious program clauses to the user. Our toolset encompasses two different techniques: ProFL$_s$, a spectrum-based technique, and ProFL$_m$, a mutation-based technique. This paper describes our Python implementation of ProFL, which is a command-line tool, released as an open-source project on GitHub (https://github.com/geoorge1d127/ProFL). Our experimental results show ProFL is accurate at localizing faults in our benchmark programs.

## CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

## KEYWORDS

Prolog, Fault localization, Declarative programming

**ACM Reference Format:**
George Thompson and Allison K. Sullivan. 2020. ProFL: A Fault Localization Framework for Prolog. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404367

## 1 INTRODUCTION

Declarative programming languages enable developers to implement rule-based systems, e.g. fault tolerant software defined network layers [10], and reason over the designs of systems, e.g. disproving the Chord ring-maintenance protocol [14]. Declarative languages express the logic of a system's behavior without giving any control flow. Then, the execution determines how to achieve the desired behavior. While a number of declarative languages exist [4, 6, 9], the broad adoption of these languages is often hindered for two key reasons. First, declarative languages have steep learning curves since their execution environment is inherently different than commonly used imperative languages, e.g. Java and C++, which use statements to sequentially change a program's state into the desired behavior. Second, declarative languages often lack robust tool support for test automation frameworks.

Prolog is a first-order logic based declarative language that has seen a wave of new users due to its applicability to rising fields such as machine learning [2]. While Prolog is viewed as one of the go to languages for artificial intelligence systems, several studies have shown that Prolog is difficult for many programmers to learn [11, 13]. Fortunately, SWI-Prolog [12] contains a unit testing framework, plunit. Besides support for unit testing, plunit is actively developing frameworks to automatically generate tests and calculate coverage, which help developers reveal bugs. However, just finding bugs is not sufficient. The user needs to be able to fix the bug. Unfortunately, locating the faulty portion of a Prolog program is difficult. Imperative languages' sequential execution allows developers to step over changes to a program's state to locate where the execution goes wrong. This is not possible in Prolog, which has no notion of control flow. Furthermore, Prolog programs can have numerous interdependent clauses spanning multiple non-contiguous lines, making the process of locating bugs time consuming.

This paper describes our Python implementation of ProFL, a toolset to perform automated fault localization for Prolog. ProFL is a command line tool that we release as an open-source project (https://github.com/geoorge1d127/ProFL). ProFL takes as input a faulty Prolog program and a corresponding plunit test suite. ProFL then provides two different types of fault localization techniques: spectrum-based techniques, which are based on well-established fault localization techniques for imperative languages [1, 5, 8], and a mutation-based technique, which is based on a recent advancement in fault localization [7]. Lastly, ProFL visualizes the results back to the user to help steam-line debugging.

## 2 BACKGROUND

ProFL integrates with SWI-Prolog, which is currently the most commonly used compilation environment for Prolog programs [12]. Figure 1 depicts a real world faulty Prolog program[1]. Prolog programs are a collection of clauses that create a knowledge base that can be queried and tested. Clauses can either be a fact or a rule and are always terminated by a period. Lines 1 - 6 depict *facts*, which use a predicate expression to make a declarative statement about the problem domain. Line 8 depicts a *rule*, which uses a predicate expression to describe relationships among facts. A rule "`A :- B`" can read as "`A` if `B`" where "`:-`" represents logical implication. Thus,

*ISSTA '20, July 18–22, 2020, Virtual Event, USA*
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3404367

[1] https://stackoverflow.com/questions/49353041/prolog-query-satisfiable-but-returns-false

```
1. male(william). /*william is male.*/
2. male(harry).
3. parent(william, diana). /*parent(x,y) - the parent of x is y.*/
4. parent(william, charles).
5. parent(harry, diana).
6. parent(harry, charles).
7. /*brother(X,Y) - the brother of X is Y.*/
8. brother(X,Y) :- X\=Y, parent(X,A), parent(Y,A), male(Y).
```

**Figure 1: Faulty Family Tree Prolog Program**

rules express ways to derive or compute new facts. On line 8, the left-hand side of the rule defines the predicate `brother` which takes as arguments two variables `X` and `Y`. The right-hand side of the rule uses negation (\), unity (=), conjunction (,) and predicates (`parent`, `male`) to express that `Y` is `X`'s brother if all of the following holds: (1) `X` cannot unify to `Y`, (2) there is a variable `A` that is a parent to both `X` and Y, and (3) `Y` is male.

Rather than sequentially executing statements, Prolog's execution environment tries to prove that a query, a user-specified sequence of predicates, is true using the defined facts and rules. plunit, Prolog's unit testing library, defines a unit test to be Prolog queries that are expected to be true for a given program. Below is a set of tests which reveal the fault on line 8 in Figure 1:

```
:- begin_tests(family).
:- include(family).
test(test1) :- brother(william,harry). /*Test Passes.*/
test(test2) :- brother(william,X). /*Test Fails.*/
:- end_tests(family).
```

The error arises because the negation of unity ("\=") is used as if it asserts the two variables must be different, which is not how the constraint gets interpreted. When Prolog resolves "`X \= Y`," the execution environment asks "Can `X` and `Y` *ever* be unified? If so, fail." This incorrect usage does not impact `test1`'s execution as both arguments to `brother` are bound to different constants and therefore cannot unify. In contrast, for `test2`, one of the arguments is a constant (`william`) and the other is an unbound variable (`X`). Since none of the clauses prevent `X` from being bound to `william`, the two can unify resulting in `test2` failing. What the user really wanted to constrain is "in any solution, `X` and `Y` must be different." In Prolog, this can be achieved by making the following change:

```
8. brother(X,Y) :- dif(X,Y), parent(X,A), parent(Y,A), male(Y).
```

The predicate `dif` is a constraint that is true if and only if `X` and `Y` are different terms.

## 3 TECHNIQUE

ProFL adapts well-studied imperative fault localization techniques to Prolog's declarative environment. Figure 2 gives an overview of ProFL's components. ProFL takes as input (1) a faulty Prolog program and (2) a plunit test suite. Then, depending on the user's selection, ProFL will run ProFL$_s$, which performs spectrum-based fault localization (SPFL), and/or ProFL$_m$, which performs mutation-based fault localization (MBFL). As output, for each fault localization technique run, ProFL displays the likely faulty clauses from most to least suspicious in separate tables. We next describe how the four main components of ProFL work.

### 3.1 Coverage Engine

Both SBFL and MBFL rely on accurate coverage information to be effective. In Prolog's execution, all clauses form one knowledge

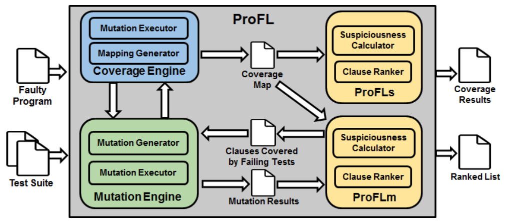

**Figure 2: ProFL Component Diagram**

base that is leveraged to try and prove a query true. To provide effective localization, we want to consider a clause covered by a test if that clause is *actively* used to resolve the test's query. Therefore, we want something analogous to statement coverage for imperative programs. Unfortunately, plunit's coverage framework is under development and does not provide the fine-grained, clause level data ProFL needs. Therefore, ProFL implements a recently developed coverage technique which utilizes mutants to calculate clause coverage [3]. To start, the mutation engine (Section 3.2) is invoked to produce all valid mutants of the program. Then, the coverage engine performs a streamlined version of mutation testing. For every mutant $m$, the test suite is executed. If a test $t$ kills $m$, then $t$'s pass/fail result changed compared to the original program. Since $m$ is a mutation of some clause $c$, we conclude $t$ covers $c$. As an optimization, once the coverage engine has detected $t$ covers $c$, $t$ is not executed for any remaining mutants of $c$. Using this process, the coverage engine builds a map from every clause to the test case(s) that cover it. This map is then passed as input to both ProFL$_s$ and ProFL$_m$. The coverage engine is intentionally kept modular to allow for the integration of other suitable coverage techniques.

### 3.2 Mutation Engine

Our mutation engine implements the following mutation operators: removing predicates, interchanging disjunction and conjunction, changing atoms or variables to anonymous variables, and interchanging arithmetic operators. When mutating a Prolog program, it is possible to accidentally create non-deterministic predicates which leads to an infinite loop during Prolog's execution. Therefore, we follow the results from a recent empirical study and avoid two types of mutants, clause reversal and cut transformation, known to create non-deterministic predicates [3]. For the coverage engine, the mutation engine generates all possible non-equivalent mutants for each clause in the program but does not control execution. For ProFL$_m$, the mutation engine generates all possible non-equivalent mutants for each clause covered by some failing test. Then, the mutation engine executes each mutant $m$ against the original test suite. As it executes, the mutation engine builds a mapping from $m$ to the tests that pass $m$ and the tests that fail $m$. Unlike the coverage engine, the mutation engine always executes the entire test suite against each mutant, as ProFL$_m$ needs detailed information about how test results change from the original to the mutated program.

### 3.3 ProFL$_s$: Spectrum-Based Fault Localization

ProFL allows users to perform traditional SBFL [1, 5, 8]. To evaluate how likely a program statement is to be faulty, SBFL utilizes test cases' pass/fail results and information about what portions of the program those test cases execute to calculate a suspiciousness

| Name | Formula |
|---|---|
| tarantula [5] | $\frac{\frac{failed(c)}{totalfailed}}{\frac{failed(c)}{totalfailed}+\frac{passed(c)}{totalpassed}}$ |
| ochiai [1] | $\frac{failed(c)}{\sqrt{totalfailed \times (failed(c)+passed(c))}}$ |
| op2 [8] | $failed(c) - \frac{passed(c)}{totalpassed+1}$ |

*totalfailed*: total number of test cases that failed.
*totalpassed*: total number of test cases that pass.
*failed(c)*: number of failed test cases that cover *c*.
*passed(c)*: number of passed test cases that cover *c*.

**Figure 3: ProFL$_s$ Supported Suspiciousness Formulas**

score for each statement. The higher a statement's suspiciousness score, the more likely the statement is to be faulty. To perform SBFL, ProFL$_s$ uses the initial test results and the coverage mapping from Section 3.1 to calculate the suspiciousness score of each clause using the three formulas outlined in Figure 3. These three formulas are commonly used in imperative SBFL techniques. *totalfailed* and *totalpassed* are the number of failed and passed test cases for the Prolog program. *failed(c)* and *passed(c)* are the number of failed and passed tests that utilize clause *c*, which is determined by our coverage framework. In general, these formulas are designed to assign a higher suspiciousness score to a clause that is covered by more failing tests and fewer, if any, passing tests. After all suspiciousness scores are calculated, ProFL$_s$ creates a ranked list of clauses from most suspicious to least, which gets displayed to the end user.

## 3.4 ProFL$_m$: Mutation-Based Fault Localization

MBFL techniques rely on tactically altering program statements and seeing how these changes affect the test results [7]. If a mutation causes a failing test to pass, then the mutated location becomes more suspicious. However, if a mutation causes a passing test to fail, then the mutated location becomes less suspicious. To perform MBFL, ProFL$_m$ extracts the clauses that are covered by failing tests from the coverage map in Section 3.1. Then, ProFL$_m$ uses the mutation engine from Section 3.2 to get a map between mutants of these clauses and which tests pass and fail each mutant. Rather than using the mutation testing information to build a mutation score for the test suite, ProFL$_m$ uses the information to help calculate a suspiciousness score for each clause using the formula in Figure 4. Within the summation, the first term correlates to the likelihood of *c* being faulty, as the term increases when mutating *c* causes a failing test to pass. The second term correlates to the likelihood of *c* not being faulty, as the term decreases when mutating *c* causes a passing test to fail. The weight $\alpha$ is based on the number of test result changes and is used to help balance the two terms, since breaking a program is easier than correcting a program. Similar to ProFL$_s$, ProFL$_m$ calculates a suspiciousness score for each clause and then sends a ranked list of suspicious clauses to be formatted and displayed to the end user.

## 4 USAGE

In this section, we describe how users can invoke ProFL. More details can be found on the ProFL GitHub homepage.

**Muse Formula [7]**

$$\frac{1}{|mut(c)|} \sum_{m \in mut(c)} \left( \frac{f_P(c) \cap p_m}{f_P} - \alpha * \frac{p_P(c) \cap f_m}{p_P} \right)$$

$$\text{where } \alpha = \frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f}$$

*$f_P(c)$*: set of test cases that cover *c* and fail *P*.
*$p_P(c)$*: set of test cases that cover *c* and pass *P*.
*mut(c)*: set of all mutants of *P* that mutates *c* and change a test result.
*$f_m$*: set of failing test for mutant *m*.
*$p_m$*: set of passing test for mutant *m*.
*mut(P)*: number of mutants created from program *P*
*f2p*: number of tests that change from failing to passing.
*p2f*: number of tests that change from passing to failing.

**Figure 4: ProFL$_m$ Supported Suspiciousness Formula**

To localize a fault, run: `python ProFL.py -p <arg> -t <arg> -f <arg> -v <arg> [-s <arg>] [-r <arg>] [-c <arg>]` or `python ProFL.py --program-path <arg> --test-suite <arg> --fl-technique <arg> --view <arg> [--suspicious-formula <arg>] [--result-path <arg>] [--coverage-path <arg>]`

- "`-p,--program-path`": This argument is *required*. Pass the file name of the faulty Prolog program.
- "`-t,--test-suite`": This argument is *required*. Pass the file name of the plunit test suite.
- "`-f,--fl-technique`": This argument is *required*. Pass the fault localization technique to use. The value should be "`-spectrum`", "`-mutation`", or "`-both`".
- "`-v,--view`": This argument is *required*. Pass how much of the ranked suspicious list to view. The value should be "`-top1`", "`-top5`", "`-top10`", or "`-all`".
- "`-s,--suspicious-formula`": This argument is *optional* and is used when the technique is "`-spectrum`" or "`-both`". Pass the suspiciousness formula for ProFL$_s$ to use. The value should be "`-tarantula`", "`-ochiai`", or "`-op2`". If specifying more than one, separate with a comma. If not specified, all three are used.
- "`-r,--result-path`": This argument is *optional*. Pass the path to which you want to save the fault localization results. If not specified, the results are only printed to the terminal.
- "`-c,--coverage-path`": This argument is *optional*. Pass the path to which you want to save the coverage results.If not specified, the coverage information is not saved.

The ProFL tool reports one table per suspiciousness formula executed. Depending on the user's selection, this table will display the most suspicious clause, the top 5 suspicious clauses, the top 10 suspicious clauses, or all ranked clauses. For each suspicious clause *c*, ProFL reports: (1) the number of test that *c* fails, (2) the number of test that *c* passes, and (3) the suspiciousness score of *c*. The user can also elect to save the fault localization and coverage results.

## 5 EVALUATION

Table 1 shows the 10 Prolog programs used to evaluate ProFL and the corresponding performance of ProFL. These programs are incorrect submissions to Exercism's Prolog exercises track[2]. **Program**

[2]https://exercism.io/tracks/prolog/exercises

**Table 1: ProFL Performance Results**

| Program | Program Details | | | Time (s) | | Ranking | | | |
|---|---|---|---|---|---|---|---|---|---|
| | # Pred | # Cls | # Tests | ProFL$_s$ | ProFL$_m$ | Tarantula | Ochiai | Op2 | Muse |
| anagram | 6 | 7 | 18 | **83.2** | 89.9 | **1** | **1** | **1** | **1** |
| binary | 5 | 3 | 16 | N/A | N/A | N/A | N/A | N/A | N/A |
| complex_num | 8 | 8 | 27 | **239.6** | 246.5 | **1** | **1** | **1** | 2 |
| dominoes | 5 | 7 | 12 | **39.8** | 69.0 | **1** | **1** | **1** | 2 |
| grains | 2 | 5 | 12 | **49.8** | 66.9 | 2 | **1** | **1** | **1** |
| hamming | 4 | 3 | 15 | **33.1** | 48.5 | **1** | **1** | **1** | 2 |
| pascal_tri | 6 | 6 | 8 | **23.1** | 30.3 | **1** | **1** | 2 | 6 |
| queen_attack | 5 | 5 | 13 | **68.3** | 80.9 | **1** | **1** | **1** | 2 |
| space_age | 3 | 10 | 8 | **23.0** | 32.2 | 9 | 9 | 9 | **1** |
| triangle | 2 | 6 | 19 | **84.4** | 143.9 | **1** | **1** | **1** | 2 |

is the name of the program: **anagram** determines which words are anagrams of a given word, **binary** converts a binary number to a decimal number, **complex_num** implements complex numbers, **dominoes** makes a chain of dominoes, **grains** calculates the number of grains of wheat on a chessboard when the number on each square doubles, **hamming** calculates the Hamming difference between two DNA strands, **pascal_tri** computes Pascal's triangle, **queen_attack** determines if two queens can attack each other on a chess board, **space_age** calculates how old someone is in terms of a planet's solar years, and **triangle** determines the type of a triangle.

The next three columns in Table 1 describe the size and complexity of the programs. **# Pred** is the number of unique predicates, **# Cls** is the number of clauses and **# Tests** is the size of the plunit test suite. We use the test suites released by Exercism for each exercise. The next two columns represent the runtime for ProFL$_s$ and ProFL$_m$ respectively. The times are in seconds and captures the time it takes to go from processing the input to displaying the results to the user. The last four columns represents the ranking given to the faulty clause by each suspiciousness formula support by ProFL: **Tarantula**, **Ochiai** and **Op2** for ProFL$_s$ and **Muse** for ProFL$_m$. The lower this value is, the better the fault localization technique performed. A value of 1 means the faulty statement was ranked as the most suspicious clause while a value of 6 means the faulty clause was ranked as the 6th most suspicious clause.

For one program, **binary**, ProFL is unable to perform fault localization as our technique got stuck in an infinite loop when mutating the faulty statement during the initial coverage calculations. This motivates our goal to incorporate additional coverage options in future releases of the tool. For the remaining 9 models, the best performing results are presented in green for both runtime and ranking. Across all programs, ProFL$_s$ runs faster then ProFL$_m$ for an average speed up of 1.4×. This is expected as ProFL$_m$'s suspiciousness formula requires targeted mutation testing in addition to the initial coverage results while ProFL$_s$'s suspiciousness formula works just with the initial coverage results. For ProFL$_s$, all three suspiciousness formulas have comparable performance with Ochiai marginally performing the best. In comparison, ProFL$_m$'s Muse formula performs slightly worse than the ProFL$_s$'s formulas, often ranking the faulty statement in an equal position (3 of 9) or one position lower (5 of 9). However, for **space_age**, ProFL$_m$ significantly outperforms ProFL$_s$. For this model, the faulty clause causes almost all tests to fail, impeding ProFL$_s$'s performance. This highlights that ProFL$_m$ can perform well in situations where ProFL$_s$ is unable to be effective. Therefore, given that the same initial coverage information can be re-used between ProFL$_s$ and ProFL$_m$, user can increase their confidence by running both techniques and leverage their individual strengths, while incurring the modest overhead from the additional mutation testing needed for ProFL$_m$. Overall, these results show ProFL can help locate faults and our two techniques have complementary roles to each other.

## 6 CONCLUSION

This paper introduced the open-source ProFL tool for automated fault localization of Prolog programs. ProFL provides command-line options to automatically perform any combination of spectrum-based or mutation-based fault localization. Given a faulty Prolog program and a fault-revealing test suite, ProFL is able to report a list of suspicious clauses for the user to investigate. Our experiments show ProFL has a minor overhead and promising accuracy results.

## REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *JSS* (2009).

[2] Ivan Bratko. 2001. *Prolog programming for artificial intelligence.* Pearson education.

[3] Alexandros Efremidis, Joshua Schmidt, Sebastian Krings, and Philipp Körner. 2018. Measuring coverage of prolog programs using mutation testing. In *International Workshop on Functional and Constraint Logic Programming.* Springer, 39–55.

[4] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM TOSEM* (2002).

[5] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *ASE.*

[6] Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)* (2010).

[7] S. Moon, Y. Kim, M. Kim, and S. Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *ICST.*

[8] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *TSE* (2011).

[9] Philip J. Pratt and Mary Z. Last. 2008. *A Guide to SQL* (8th ed.). Course Technology Press.

[10] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. 2013. FatTire: Declarative Fault Tolerance for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.* 109–114.

[11] Maarten W. Van Someren. 1990. What's wrong? Understanding beginners' problems with Prolog. *Instructional Science* 19, 4/5 (1990), 257–282.

[12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[13] Shanshan Yang and Mike Joy. 2007. Approaches for Learning Prolog Programming. *Innovation in Teaching and Learning in Information and Computer Sciences* 6, 4 (2007), 88–107.

[14] Pamela Zave. 2012. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.* 42 (2012), 49–57.