ProFL: A Fault Localization Framework for Prolog

George Thompson
North Carolina A&T State University, USA
gmthompson11@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: ProFLc, a spectrum-based technique, and ProFLm, 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:

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 program1. 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,

1https://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:

1. :- begin_tests(family).
2. :- include(family).
3. test(test1) :- brother(william,harry). /*Test Passes.*/
4. test(test2) :- brother(william,X). /*Test Fails.*/
5. :- 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 ProFLs, which performs spectrum-based fault localization (SPFL), and/or ProFLm, 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 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 ProFLs and ProFLm. 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 ProFLm, 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 ProFLm needs detailed information about how test results change from the original to the mutated program.

3.3 ProFLs: 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
within the summation, the first term correlates to the likelihood
of test that covers 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 creates a ranked list of clauses from
most suspicious to least, which gets displayed to the end user.

### 3.4 ProFL<sub>m</sub>: 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<sub>m</sub> extracts the clauses that are covered by failing tests from the coverage map in Section 3.1. Then, ProFL<sub>m</sub> 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<sub>m</sub> 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<sub>s</sub>, ProFL<sub>m</sub> 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.

### Supported Suspiciousness Formulas

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>tarantula</td>
<td>[ \frac{\text{failed}(c)}{\text{totalpassed}} ] \times \frac{\text{failed}(\text{mut}(c)) \cap \text{mut}(\text{passed}(c))}{\text{mut}(\text{passed}(c))} ]</td>
</tr>
<tr>
<td>ochiai</td>
<td>[ \sqrt{\frac{\text{totalpassed} \times \text{failed}(\text{mut}(c))}{\text{passed}(\text{mut}(c)) \times \text{failed}(c)}} ]</td>
</tr>
<tr>
<td>op2</td>
<td>[ \text{failed}(c) - \frac{\text{passed}(c)}{\text{totalpassed}} ]</td>
</tr>
</tbody>
</table>

**Figure 3: ProFL<sub>s</sub>, 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<sub>s</sub> 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<sub>s</sub> creates a ranked list of clauses from
most suspicious to least, which gets displayed to the end user.

### 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 track2. Program

![Figure 4: ProFL<sub>m</sub>, Supported Suspiciousness Formula](https://exercism.io/tracks/prolog/exercises)

To localize a fault, run:

```python
python ProFL.py -p <arg> -t <arg>
```

- **--program-path <arg>** specifies the name of the faulty Prolog program.
- **--test-suite <arg>** specifies the name of the plunit test suite.
- **--fl-technique** specifies the fault localization technique 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.
- **--suspicious-formula** specifies how much of the ranked suspicious list to view. The value should be `"top1"`, `"top5"`, `"top10"`, or `"all"`. If not specified, all are used.
- **--coverage-path <arg>** specifies which coverage results to use. The value should be `"p2f"` or `"f2p"`. 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
suspicous 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.

---

1. https://exercism.io/tracks/prolog/exercises
2. Program
Table 1: ProFL Performance Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Program Details</th>
<th>Time (s)</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Pred # Cls # Tests</td>
<td>ProFLm ProFLm</td>
<td>Tarantula Ochiai Op2 Muse</td>
</tr>
<tr>
<td>anagram</td>
<td>6 7 18</td>
<td>83.2 89.9</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>binary</td>
<td>5 5 16</td>
<td>N/A N/A</td>
<td>N/A N/A N/A N/A</td>
</tr>
<tr>
<td>complex_num</td>
<td>8 8 27</td>
<td>239.6 246.5</td>
<td>8 1 1 1 2</td>
</tr>
<tr>
<td>dominoes</td>
<td>5 7 12</td>
<td>39.8 69.0</td>
<td>3 1 1 2</td>
</tr>
<tr>
<td>grains</td>
<td>2 5 12</td>
<td>49.8 66.9</td>
<td>2 1 1 1</td>
</tr>
<tr>
<td>hamming</td>
<td>4 3 15</td>
<td>33.1 48.5</td>
<td>1 1 1 2</td>
</tr>
<tr>
<td>pascal_tri</td>
<td>6 6 8</td>
<td>23.1 30.3</td>
<td>1 1 2 6</td>
</tr>
<tr>
<td>queen_attack</td>
<td>5 5 13</td>
<td>68.3 80.9</td>
<td>1 1 1 2</td>
</tr>
<tr>
<td>space_age</td>
<td>3 10 8</td>
<td>23.0 32.2</td>
<td>9 9 9 1</td>
</tr>
<tr>
<td>triangle</td>
<td>2 6 19</td>
<td>84.4 143.9</td>
<td>4 1 1 2</td>
</tr>
</tbody>
</table>

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 ProFLm and ProFLm 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 and Muse for ProFLm. 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, ProFLm runs faster than ProFLm for an average speed up of 1.4×. This is expected as ProFLm’s suspiciousness formula requires targeted mutation testing in addition to the initial coverage results while ProFL’s suspiciousness formula works just with the initial coverage results. For ProFLm, all three suspiciousness formulas have comparable performance with Ochiai marginal performing the best. In comparison, ProFLm’s Muse formula performs slightly worse than the ProFL’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, ProFLm significantly outperforms ProFL. For this model, the faulty clause causes almost all tests to fail, impeding ProFL’s performance. This highlights that ProFLm can perform well in situations where ProFL is unable to be effective. Therefore, given that the same initial coverage information can be re-used between ProFL and ProFLm, 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 ProFLm. 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