Mutating Sample Solutions to Improve Prolog Exercise Tasks and Their Test Suites

Ivan Khu Tanujaya, Janis Voigtländer[1], Oliver Westphal[2]

Abstract: We report on a tool for analyzing and adapting test suites for Prolog in an educational setting. The key idea is to mutate a known-correct sample solution in order to "simulate" possible student mistakes, and use that to "harden" the test suite that an e-learning system uses for judging submissions and giving feedback.

Keywords: Prolog; Teaching; Testing

1 Introduction

At our department, Prolog is taught in two courses with non-overlapping audiences, including one where it precedes the introductory programming course. In both cases, practical exercises play an important role in the teaching and self-study. That is, students are given weekly Prolog modeling/programming tasks which they are supposed to solve and submit to an online tutoring system. The system gives immediate feedback about correctness of the submission and potentially information about program misbehavior, helping the students to improve their solution attempts step by step. The feedback functionality depends on testing, and thus, on appropriate test suites written by the instructors beforehand. It turns out that assembling appropriate collections of test cases can be challenging. Even after several semesters and iterations with similar or even the same exercise tasks, we still encounter student submissions that are accidentally mis-characterized as correct by the system (more seldom the opposite situation), or where the feedback is not as helpful as we would hope. So, to help us in the role of instructors, a tool was developed that assists in test suite construction, based on mutation testing [DLS78, JH11]. The tool comes with a parameterizable and extensible catalog of mutations, drawing on earlier work [Ef18, TV06], but trying to put emphasis on (new) mutations effective in our educational application, as well as a GUI that supports a workflow we found useful. In this article, we report on the rationale, realization, and experiences.

[1] University of Duisburg-Essen, Faculty of Engineering, Germany, janis.voigtlaender@uni-due.de
[2] University of Duisburg-Essen, Faculty of Engineering, Germany, oliver.westphal@uni-due.de

2 Practical Setting

Let us first describe the setup we are operating in, since its specifics also constrain what kind of tests we can include in the test suite etc. As e-learning platform for the online exercises we use Autotool. Some aspects of it have been described in earlier works, with views towards "theory tasks" [Wa17] and "Haskell programming tasks" [SVW19]. For Prolog programming tasks, it is equipped with a small Prolog interpreter[3] that supports only the sublanguage that we actually need in the courses, but has additional features like the possibility to display derivation trees (as feedback to students) of queries for a program. Specification of an exercise task by an instructor consists of a verbal description, possibly some background facts and rules provided to students, and a test suite kept hidden originally (but potentially revealed incrementally to students as they submit solution attempts).

For example, a typical task could provide a family tree fact base like the following:

```
female(anna). female(juliet). ...
male(harry). male(luke). ...
child(lisa,anna). child(mary,juliet). ...
```

Then, students may be asked to implement certain predicates like `brother/2`. And the e-learning system could use tests listing the full extensions of these predicates:

```
brother(X,Y): brother(peter,mary), brother(peter,luke), ...
```

Or we could specify several different tests per predicate, for example with variables partially instantiated, if desired:

```
brother(peter,Y): brother(peter,mary), brother(peter,luke), ...
brother(X,luke): brother(peter,luke), ...
```

For any student submission, the query `brother(X,Y)` – or other kinds of tests, mentioned above and below – would then be run against the known fact base and the results compared to what the test listing says. Feedback is tuned to not make it overly easy to circumvent the task's intent. For example, here the correct extension would not be displayed to students. If they submit a not yet correct definition for `brother/2`, they would be told what instances *their* predicate produces, and that this list differs from the correct one, but not be told what the correct one is exactly. Of course, they could still work out from the family tree itself what all brothers are and of whom, and simply submit a program consisting of facts like `brother(peter,mary)`, instead of a general rule `brother(X,Y) :- male(X),child(X,Z),...`, but we have ways of navigating around this (e.g., having some hidden family members to be used in some tests, and that the students do not know about from the public fact base).

[3] `https://hackage.haskell.org/package/prolog`, `https://github.com/fmidue/prolog`

Another example task is as follows:

```
/* Write a predicate g/3 that, when interpreted as a function, inserts
 * some specific element between each two adjacent elements of a given
 * list. For example, g(a,[b,a,d],[b,a,a,a,d]) should hold.
 */
```

The test suite in this case consists not of predicate extension lists, but simply several queries that are expected to be true. One of them is the test already mentioned in the task description, others are negative tests like `not(g(a,[],[_]))`, and yet others use much longer lists that are generated using auxiliary predicates not shown to students. Individual tests can be marked as hidden (so that students will only be told that a test failed but not what was queried), can be annotated with verbal descriptions to be used in the feedback (e.g., so that failure of a hidden test can still convey some information to students), can be marked as derivation-tree-producing, and can be equipped with timeouts.

The features mentioned above generally allow us to write test suites that catch wrong submissions and give useful feedback to students, on the kind of exercise tasks we typically pose. Some of the features (like hidden predicates and hidden tests) try to compensate for lack of a "proper" test specification language including generators for random test data in the tradition of QuickCheck [CH00, AFC14]. One could argue that we should invest into such more powerful approaches once and for all, but realistically there are factors preventing this, such as a certain lock-in to what we already have and use, our decision to work with an own mini-Prolog implementation (that allows us to display the lecture's flavor of derivation trees) rather than something full-fledged, but also plain and simple the limited amount of resources we can pour into this part of our teaching activities. So, what we report on here can also be seen as the result of doubling down on, and pragmatically embracing, the situation we have, confining ourselves to the kind of test suites described above, but exploring what we can do in terms of growing and improving specific instances by following this simple idea: "For a given task, take a sample solution, mutate it in many ways, run the outcomes against the existing test suite, observe whether/where tests are lacking, and if so, modify/add, then repeat."

3 Relation to Existing Work

Several related works have already been mentioned throughout the preceding two sections, and of course, there is much more on mutation testing generally. Earlier works in the Prolog context [Ef18, TV06] initially inspired which mutation operators to perform. But our educational setting comes with certain particularities that to some extent change the game.

For one thing, our exercise tasks are typically quite small. Sample solutions most often span only about a handful of lines, and test suites often between a dozen and a score. That leads to a completely different situation than in applications of mutation testing in software

engineering scenarios, where a code base several orders of magnitudes larger is to be tested against an also much larger test suite. There, the notion of kill ratio is of interest, since one is concerned with coverage questions and reaching 100 % is rather illusionary. In our setting, instead, we are interested in a binary decision. Put differently, we do not want *any* (reasonably complex, realistically written by students) wrong program to pass the test suite; killing some non-100 % ratio of them is not satisfactory.[4] Another consequence of the small program sizes is that we can realistically go for exhaustiveness on the combinatoric choices. We can, in principle, produce and test *all* programs obtained by applying mutation at one or several places in the code. For code repositories found in software engineering scenarios, that would be prohibitively expensive.

Another particularity comes from the kind of mutations to consider. Put simply, beginner programmers tend to make different (or at least additional) mistakes than software engineers. Or, put even more bluntly, the "competent programmer hypothesis" that is often taken to underlie the mutation testing approach just does not apply. Consequently, new mutation operators become relevant. That is why it was important to us that the developed tool does not just come with a catalog of preexisting mutation operators, but is extensible in a rather ad-hoc way, possibly even from week to week as we make ongoing observations about typical (as well as surprising/untypical) student mistakes in exercises for the course or special portions of it.

The last mentioned point can even go so far that we will employ mutations that conceivably nobody outside our own circle would contemplate. In one of our two Prolog-related courses, the Prolog part comes after a part teaching Haskell. Hence, some students tend to confuse aspects of the two languages for a while, trying to use Haskell syntax or idioms in Prolog exercises.

For example, a student might want to implement a predicate equivalent to a Haskell function `p :: [a] -> Bool`. Wrongly assuming Prolog is typed somehow, or at least attempting to bring the "`[a]`" piece from the Haskell declaration line into play on the Prolog side as well, they write a rule `p([A]) :- ...`, essentially trying to indicate the typing that way. Now this rule for `p` matches only singleton lists, which is most likely not the intended behavior. Sometimes a single well-chosen test case can immediately, and informatively, give feedback that dispels such misconception. But to include that test case up front, we have to anticipate that students might make the specific kind of mistake in the current exercise. By modeling such mistakes via mutation operators, and developing a habit of checking our test suites through the mutation tool, we can never again forget to cater for them. But for general software engineering practice, even in the Prolog context, these specific mutation operators would likely not be of interest at all.

[4] Which is not to say that we can guarantee we will always reach 100 %. But the aspiration is different from the outset. Yet still realistic, given the constrained nature of cases.

4 What Was Implemented

The tool consists of a frontend written in JavaScript/React and a backend written in Haskell. Both are available on GitHub:

- `https://github.com/fmidue/prolog-mutator`
- `https://github.com/fmidue/prolog-test-server`

Some existing mutations are implemented on the JavaScript side, some on the Haskell side, where most likely also future extensions of the mutation catalog will reside. Other duties of the Haskell backend are to parse Prolog source code in service to the JavaScript implemented mutations, and to run Prolog sample solutions and mutations thereof against the test suites, using the mini-Prolog interpreter also employed when evaluating actual student submissions. Indeed, as much as possible code is shared between the Haskell backend here and the Prolog task type in the e-learning platform Autotool (which is also implemented in Haskell).

When starting the workflow of improving an exercise task, the existing task configuration (including the test suite) and a sample solution are loaded and run through the interpreter, see Fig. 1.

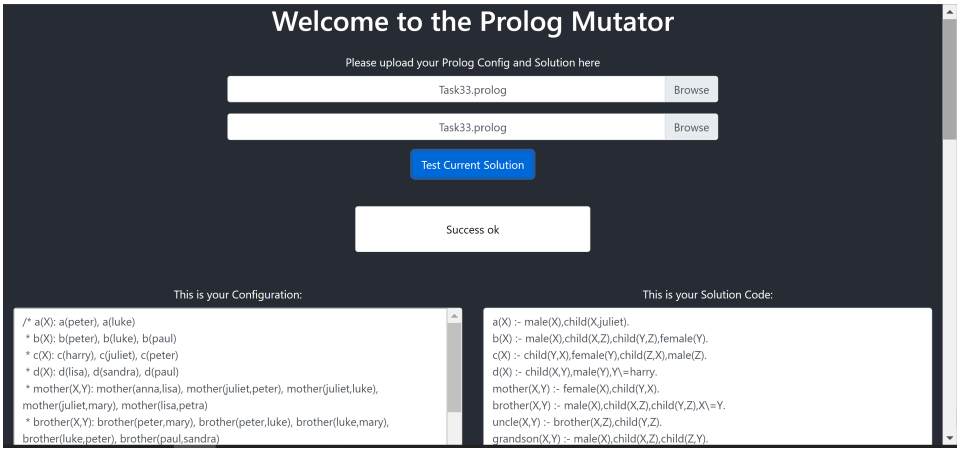

Fig. 1: Start of our workflow

Then, one makes selections from a catalog of mutations offered, see Fig. 2 for an excerpt. Mutation options labeled as `[Individual]` change exactly one place in the Prolog code per run, whereas ones labeled as `[Summary]` change several places at once. As additional parameter, the user sets the number of mutants to create per selected option. So, despite our take on exhaustiveness, it is still possible to impose limits for practical reasons. Nondeterminism comes into play then, as several runs with the same parameter settings can lead to different specific mutant sets being created.

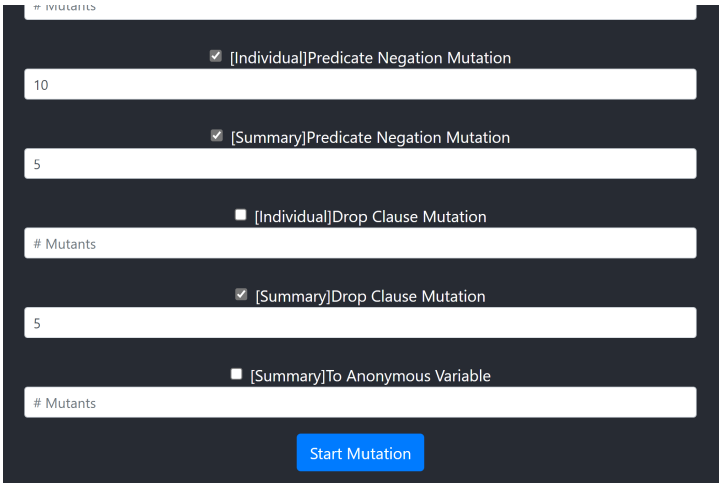

Fig. 2: The mutation catalog and setting of parameters

After each run (on demand, repeatedly) with the selected options, we get a table listing the mutants, how they were obtained and whether or not they were killed by the test suite, see Fig. 3.

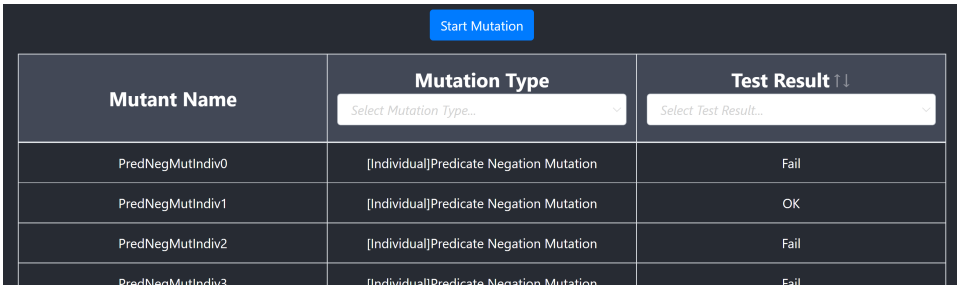

Fig. 3: Report list of created mutants

We can filter that table, for example focusing on the mutants that survived the test suite, in order to investigate whether that survival is problematic. We can also take a closer look at each separate mutant, see Fig. 4, and even change it further by hand and re-test in order to see what effect that has. Of course, we can also go back to the top of the page (Fig. 1), and informed by what we learned from investigating one or more mutants, edit the test suite directly in the left pane, re-test existing mutants with the new test suite, and if we want to, start a new iteration in our overall process, with fresh mutants etc.

The above closes the description of our workflow. The specific mutation operators initially on offer are described in [Ta21]. Future extension will focus on operators that mimic student

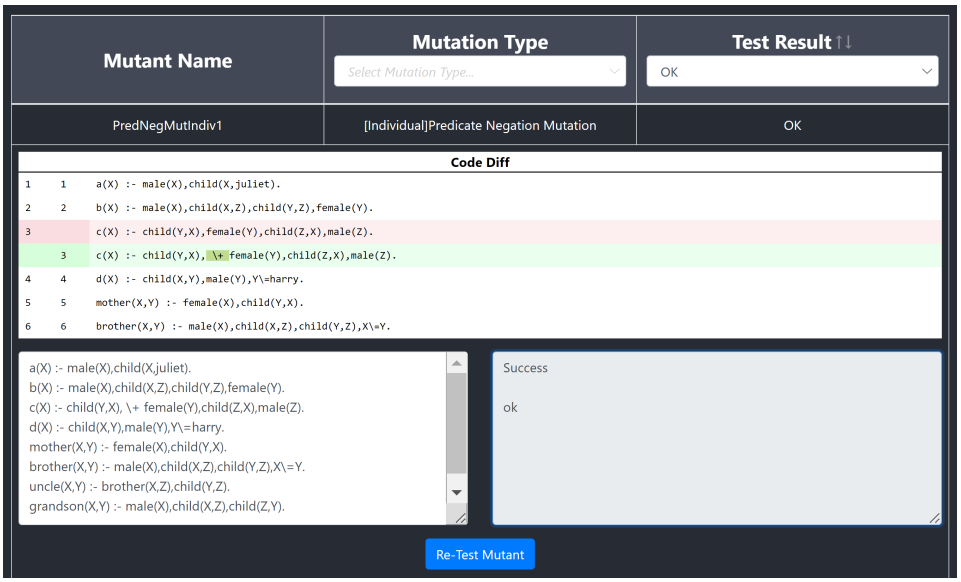

Fig. 4: Investigating a mutant that survived

mistakes we encounter in practice (e.g., erroneously writing a variable `A`, that happens to be intended to be a list, as `[A]`, as described above).

5 What We Learned

Already, we had opportunity and occasion to reconsider/improve several of real exercise tasks in use. As a brief report on some cases:

- In the example demonstrated through Figs. 1 to 4, the remedy was actually not a change to the test suite. Indeed, the mutant we found there demonstrated a defect in the fact base given as part of the task. Namely, while the subtask for predicate `c/1` was to query for “persons with a daughter and a son”, it so happened that in the specific family tree provided there were no persons with a son but no daughter. That is, every son had a sister, so even just querying for “persons with a son” was equivalent to “persons with a daughter and a son”. The fix will be to alter the family tree in the next installment of the exercise.
- In our setting we found mutation testing to be very effective for ensuring that we have (enough) negative test cases. Using too few negative test cases is apparently particularly likely to happen when creating new Prolog tasks from scratch after having created a few Haskell tasks (or when transferring Haskell tasks to Prolog, as we sometimes do), because in Haskell negative test cases are often less important

due to the functional instead of relational semantics of code. In one case, exploring mutations revealed a deficit in negative tests in the suite for a task we had been using and maintaining for several years. The task ostensibly required a predicates' output argument to be a list with a very particular order of elements based on the inputs, but without a certain negative test it actually allowed much more unspecified contents as long as the length of the list was correct.

- Due to the setup of the Prolog task type in Autotool, which always copies and prepends the task template to anything a student submits, some unintended student edits to that code can be inadvertently shadowed. For example, in a cryptarithmetic puzzle task (teaching generate-and-test), the task template contained the rule

```
solve(A,B,C,D,E,F,G,H,I) :-
  generate(A,B,C,D,E,F,G,H,I),
  test(A,B,C,D,E,F,G,H,I).
```

  and students were asked to implement the `generate` and `test` predicates. However, not all students stick to that, instead also touching the implementation of `solve` itself, thus defeating the learning purpose of the task. But that deviation is not necessarily detected by the test suite since Autotool simply preserves the original definition of `solve` from above as well, thus bringing it in competition (Prolog nondeterminism style) with the student's version. Mutation testing at least sensitized us to that for this task, a possible remedy being not to have the `solve` implementation given in the template verbatim up front.

- In the same cryptarithmetic puzzle task as mentioned above, some at first glance strange mutants for arithmetic expressions were not killed. The explanation then was actually simple: solutions of the puzzle had one variable/number be zero, so of course it did not matter whether terms involving that variable as a factor were added or were subtracted.

6 Conclusion

We have used the tool for a while now and seen its benefits. Application so far has been on checking and improving existing test suites. However, we also envision further uses surrounding the tool.

For example, we might want to use it for new exercise tasks created from scratch, that is, with an empty test suite at first. In principle, it is indeed possible to start without any tests. That means that initially all mutants will pass, which is not very useful information to guide test writing. To get a more productive start, the tool could be extended to create some initial test suite from the given sample solution. After all, we have a Prolog interpreter at hand and under our control, so if we want test cases for a predicate, say `g/3`, we can issue a query `g(X,Y,Z)`, collect some output assignments for the variables, and turn these assignments

into positive test cases. After that, mutation testing can kick in to guide us towards further, also negative, test cases.

Another possible use is to employ mutation testing directly for feedback generation. That is, instead of mutating our own sample solution (with the aim of producing a better test suite and thus eventually appropriate feedback to students), we would mutate student submissions and report findings from that. For example, the following is a submission we got in a recently running course for the `g/3` task from Sect. 2:

```
g(_, Y, Y) :- Y = [_ | []]; Y = [].
g(X, [A, B | C], [A, X, B | D]) :- g(X, [B | C], [B | D]).
g(X, [A, B], [A, X, B]) :- g(X, [], []).
```

It is a submission we would mark as a correct solution, and indeed it passes our current test suite. However, it would also do so if the last line would be dropped. The "drop clause mutation" would tell us as much, and could thus be employed to report to the student that the submitted program is unnecessarily complicated: it contains a redundant rule.

As a contra point to our focus on test suites produced by the lecturer, it would also be interesting to consider test activities that students should ideally undertake themselves, as in related work [Ch04, Ed03]. Anecdotally, we can confirm the observation that students tend to do less testing on their own when there is a system/grader to ultimately rely on for each submission. To provoke active test writing, the tables could be turned for some exercises: students are given a program behavior description as usual, but instead of being asked to write a predicate implementing the behavior are asked to write a test suite that successfully distinguishes good (correct program) from bad (mutants to be killed).

Acknowledgments. We would like to thank the reviewers for their comments and suggestions, as well as ideas for future work.

Bibliography

[AFC14] Amaral, C.; Florido, M.; Costa, V.S.: PrologCheck – Property-Based Testing in Prolog. In (Codish, M.; Sumii, E., eds): 12th International Symposium on Functional and Logic Programming, FLOPS 2014, Proceedings. volume 8475 of Lecture Notes in Computer Science. Springer, pp. 1–17, 2014.

[CH00] Claessen, K.; Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In (Odersky, M.; Wadler, P., eds): Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, Proceedings. ACM, pp. 268–279, 2000.

[Ch04] Chen, P.M.: An automated feedback system for computer organization projects. IEEE Trans. Educ., 47(2):232–240, 2004.

[DLS78] DeMillo, R.A.; Lipton, R.J.; Sayward, F.G.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer, 11(4):34–41, 1978.

[Ed03] Edwards, S.H.: Improving student performance by evaluating how well students test their own programs. ACM J. Educ. Resour. Comput., 3(3):1:1–1:24, 2003.

[Ef18] Efremidis, A.; Schmidt, J.; Krings, S.; Körner, P.: Measuring Coverage of Prolog Programs Using Mutation Testing. In (Silva, J., ed.): 26th International Workshop on Functional and Constraint Logic Programming, WFLP 2018, Revised Selected Papers. volume 11285 of Lecture Notes in Computer Science. Springer, pp. 39–55, 2018.

[JH11] Jia, Y.; Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, 37(5):649–678, 2011.

[SVW19] Siegburg, M.; Voigtländer, J.; Westphal, O.: Automatische Bewertung von Haskell-Programmieraufgaben. In (Strickroth, S.; Striewe, M.; Rod, O., eds): Fourth Workshop „Automatische Bewertung von Programmieraufgaben“, ABP 2019, Proceedings. GI, pp. 19–26, 2019.

[Ta21] Tanujaya, I.K.: A Tool for Mutation Testing of Prolog Exercise Tasks. Bachelor thesis, University of Duisburg-Essen, 2021. `https://github.com/fmidue/prolog-mutator/blob/main/Bachelorarbeit_Tanujaya.pdf`.

[TV06] Toaldo, J.; Vergilio, S.: Applying Mutation Testing in Prolog Programs. In: VII Workshop de Testes e Tolerância a Falhas, Proceedings. 2006.

[Wa17] Waldmann, J.: Automatische Erzeugung und Bewertung von Aufgaben zu Algorithmen und Datenstrukturen. In (S.Strickroth; Müller, O.; Striewe, M., eds): Third Workshop „Automatische Bewertung von Programmieraufgaben“, ABP 2017, Proceedings. volume 2015 of CEUR Workshop Proceedings. CEUR-WS.org, 2017.