Measuring Coverage of Prolog Programs Using Mutation Testing

Alexandros Efremidis[1], Joshua Schmidt[1] ✉ [0000−0001−8842−2993], Sebastian Krings[2] [0000−0001−6712−9798], and Philipp Körner[1] [0000−0001−7256−9560]

[1] Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
{alefr101,joshua.schmidt,p.koerner}@uni-duesseldorf.de
[2] Niederrhein University of Applied Sciences
Mönchengladbach, Germany
sebastian.krings@hs-niederrhein.de

Abstract. Testing is an important aspect in professional software development, both to avoid and identify bugs as well as to increase maintainability. However, increasing the number of tests beyond a reasonable amount hinders development progress. To decide on the completeness of a test suite, many approaches to assert test coverage have been suggested. Yet, frameworks for logic programs remain scarce.
In this paper, we introduce a framework for Prolog programs measuring test coverage using mutations. We elaborate on the main ideas of mutation testing and transfer them to logic programs. To do so, we discuss the usefulness of different mutations in the context of Prolog and empirically evaluate them in a new mutation testing framework on different examples.

1 Introduction & Motivation

Testing is an important aspect in professional software development, both to avoid and identify bugs as well as to increase maintainability. However, tests themselves again consist of source code and possibly further artifacts that need to be maintained. In modern software systems, code only needed for testing purposes can contribute between 33% and 50% to the overall source code of a project [19,5]. In consequence, increasing the number of tests beyond a reasonable amount again hinders development progress. The key to an efficient test suite is to assert the code coverage of existing tests. That means, to measure to what extent production code is tested. Afterwards, one can remove tests that do not cover additional aspects and add tests where code is uncovered. In Section 2, we present different approaches to measure code coverage.

This paper makes two contributions: Firstly, we discuss several program transformations used for mutation testing in Section 3, inspired by Toaldo and Vergilio [23], and argue whether we deem them to be sensible or not. Secondly, we provide an implementation of a framework featuring several mutations. Our framework is publicly available for SWI and SICStus Prolog and is presented in

Section 4. In Section 5, this framework is used to evaluate whether our intuitive classification of mutations as sensible or foolish is correct by measuring the test coverage of several selected examples. Finally, we discuss related and future work in Sections 6 and 7.

2 Code Coverage Metrics

Different metrics for code coverage, mostly differing in granularity, have been suggested and are at least partially applicable to Prolog:

Predicate, Clause, Sub-Goal Coverage: A simple way to gain some insight into code coverage is to execute tests and trace which code was executed. This can be done on different levels, that is, on the level of sub-goals, clauses or predicates. Moreover, there are different metrics to decide whether a specific level is covered. For instance, a sub-goal, clause or predicate can be regarded as covered if it succeeded at some point during execution. In this paper, we use a more restricted metric. We regard a clause to be covered if all sub-goals are covered, and, analogously, regard a predicate to be covered if all clauses are covered. Success on each level can be traced, for instance, by inserting tracing code via term expansion (source-to-source transformation) as done by Krings [8], using hooks into the Prolog interpreter (as done in SWI-Prolog's [24] testing framework PlUnit) or executing the code in a meta-interpreter.

Branch Coverage: Instead of considering individual program points covered if reached, branch coverage considers if branching points such as if-statements have been executed both ways. In particular, this implies that each condition has been evaluated to both true and false at least once. For Prolog, this could be implemented either on the level of conditions, but also on the level of individual calls. In this case, we would expect the test suite to make each call fail and succeed at least once. This would be harder to reach than the predicate coverage introduced above, given that each predicate would additionally have to fail at least once. In case of Prolog, one also has to decide if and how redos of predicates should be counted.

Path Coverage: Path coverage abstracts further from individual program points. Instead of enforcing each condition to be evaluated in both directions, path coverage considers all combinations of decision, that is, all paths through a predicate. As above, one has to decide how redos should be counted, that is, if each combination of redo and later succeed is an individual path.

MC/DC Analysis: A more sophisticated and popular approach is coverage analysis via MC/DC (modified condition / decision coverage). In order for code to be considered covered by MC/DC, all conditions and decisions have to take all outcomes and each condition of a decision has to independently influence the overall outcome of the decision. MC/DC analysis can be implemented via term expansion as well.

2.1 Mutation Testing

The idea behind *mutation testing* is to determine the test coverage by asserting the effectiveness of a test suite on modified versions of the source code under test. To do so, syntactically not equivalent versions of the source code, called *mutants*, are generated which are intended to be semantically not equivalent to the original code. We view two programs to be semantically not equivalent if they produce a different output for the same input at least once. For instance, a mutant is generated by replacing an equality with an inequality. Afterwards, all tests are executed.

Once a semantically not equivalent mutant is considered, we expect at least one test to change its outcome. That means, at least one positive test fails or a negative test succeeds. If this is the case, the mutant is called *dead*, indicating that the test suite covers the mutated clause. Otherwise, the mutant is called *alive*, indicating a lack of coverage.

To finally determine the coverage, a mutation testing tool will run the tests on the mutant, check the result and reset the mutant to the original code for the next iteration. This workflow continues until no further mutation is possible. Afterwards, the so-called *mutation score*, i.e., the number of dead mutants divided by the number of generated mutants, is computed. The mutation score can be used as a measure for the test coverage.

As one can see, it is crucial to compute mutants that are indeed not equivalent regarding their semantics, as a semantically equivalent mutant will always be considered alive. In consequence, the mutation score becomes less meaningful with an increasing amount of semantically equivalent mutants. However, deciding whether two programs are semantically equivalent is, in general, undecidable [18].

To counter this, it is possible to approximate the equivalence of two source code snippets by constraining the domains of used arguments, as for example suggested by Offutt and Pan [16]. Afterwards, their equivalence is checked exhaustively within these restricted domains. Of course, this might result in detecting false positives depending on the chosen domains.

In consequence, special care has to be taken when selecting mutation operators to be applied to the source code [6]. While it is only seldom possible to find mutation operators without risk of generating semantically equivalent mutants, the risk of semantically identical mutants differs between operators.

Libraries of useful mutations have been suggested for other languages, such as the Javalanche framework [20] and PIT [3] for Java, Mull [4] for LLVM, MuCheck [12] for Haskell, and many more [7]. Of course, some mutations like changing relational and arithmetic operators or constant values are similar in different programming languages. However, due to the different nature of Prolog, those libraries cannot easily be adapted to Prolog. In Prolog, a predicate is either true or false and values are input and output via its arguments. Instead of writing a sequence of instructions like in imperative programming languages, one describes the solution for a problem declarative. Prolog predicates do not determine which arguments are inputs or outputs and can possibly be used in several directions like the predicate `append/3`. If calling `append([1],[2],R)`, the two lists are concatenated and we derive `R = [1,2]`. If calling `append(A,B,[1,2])`, we derive ground lists `A` and `B` that can be concatenated to the list `[1,2]`. Here, we are able to derive all solutions via backtracking. To that effect, a mutation like changing return values as PIT suggests for Java is not applicable in Prolog by default. Besides that, there are more mutations that are specific for certain programming languages like replacing constructor calls with `null` values in Java or type-aware function replacements in Haskell.

3 Mutation Operators

In the following, we introduce our selection of mutation operators, which is based on the selection suggested by Toaldo and Vergilio [23]. We distinguish between *sensible* operators, which we expect to yield semantically different programs, and *foolish* operators, where programs are expected to retain the original semantics in most cases. For most mutations, examples that represent idiomatic Prolog code are given. In all cases, we expect existing test cases to be reasonable: for example, if the actual test initially fails, backtracking should be avoided and the test should fail. Furthermore, the test should compare with a (mostly) ground term instead of allowing unification generously. Test cases can either prove or disprove a goal. When applying mutation testing, we expect all tests to succeed. A test disproving a goal should also succeed by checking for failure.

3.1 Sensible Mutations

From our experience, we consider the following transformations to be sensible:

Predicate Removal: Deleting a predicate ϕ, more precisely all clauses of ϕ with the same arity, is a sensible mutation because at least one test should fail, otherwise ϕ is not tested at all. As long as ϕ is not dead code, the semantics change by removing ϕ due to the occurring existence error. This mutation is comparable to predicate coverage, as we expected at least one test to call ϕ.

Disjunction to Conjunction: By mutating a disjunction to a conjunction, only a subset of queries can succeed: now, they have to satisfy both branches. Similar to branch coverage, we expect tests to cover each branch individually. In particular, there should exist a test where the first alternative fails whereas the second succeeds. An example is given in Figure 1.

In propositional logic, replacing a disjunction with a conjunction does not necessarily change the semantics since a disjunction also provides the case where both arguments are true. Prolog, on the other hand, does not execute the case

```
is_empty(L) :-                  is_empty(L) :-
    L == [], ! ; fail.              L == [], ! , fail.
```

Fig. 1. Changing Semantics by Replacing a Disjunction with a Conjunction.

that both disjuncts are true. Instead, Prolog introduces a choice point leading to backtracking when searching for another solution. This choice point is not retained by the mutant. In practice, the calls within a disjunction often refer to the same variables providing alternative results. To that effect, we expect replacing a disjunction with a conjunction to alter the semantics in most cases, and, thus, to be sensible in Prolog.

Conjunction to Disjunction: Replacing a conjunction by a disjunction does not necessarily change the semantics in propositional logic. For instance, if $A \wedge B$ is true, the disjunction $A \vee B$ will be true as well. If $A \wedge B$ is false, the disjunction will be true if A or B is true, which changes the semantics for all interpretations but $A = \perp \wedge B = \perp$. In Prolog however, the data flow of a predicate often consists of passing arguments between predicates within a conjunction as can be seen in Figure 2. In this context, replacing a conjunction by a disjunction will most likely change the semantics since the execution of this predicate will terminate after executing the first argument of a disjunction but initializing a choice point. This choice point also applies if a disjunction fails. In case both arguments fail or succeed and do not pass arguments among each other or depend on any shared state, this mutation will not change the semantics like in propositional logic.

```
flatten([L|Ls], FlatL) :-         flatten([L|Ls], FlatL) :-
    flatten(L, NewL),                 flatten(L, NewL) ;
    flatten(Ls, NewLs),               flatten(Ls, NewLs) ,
    append(NewL, NewLs, FlatL).       append(NewL, NewLs, FlatL).
```

Fig. 2. Changing Semantics by Replacing a Conjunction with a Disjunction.

Atom or Variable to Anonymous Variable: Turning an atom or a variable to an anonymous variable causes that certain values are no longer ground which most likely changes the semantics. Yet, tests may still pass if recursive cases are not tested. Nevertheless, the predicate might be too complicated, taking a variable as parameter that is not used within a clause. Moreover, a variable might be a singleton one. In both cases, replacing this variable by an anonymous one will not necessarily change the semantics of this specific clause as can be seen in Figure 3. However, assuming that a clause does not define singleton or unnecessary variables, this mutation will change the semantics in most cases. These false positives still bear meaning about code quality, where singleton variables and unnecessary parameters qualify as “code smell” that should be avoided.

```
remove_dups([X,X|T],W,Res) :-       remove_dups([X,X|T],_,Res) :-
    remove_dups([X|T],W,Res).           remove_dups([X|T],_,Res).
```

Fig. 3. A Predicate Retaining its Semantics when Replacing a Variable with an Anonymous One.

Interchanging Arithmetic Operators: When replacing two arithmetic operators with each other (e.g. replacing `+` with `*`), a sensible mutant is likely to be created. Since there are many operators to choose from it is important to not create multiple mutants to avoid a disproportional impact on the overall mutation score. For instance, if an arithmetic operation is well tested, every mutation should fail. Using several mutations would lead to a higher mutation score, although the branch is already ensured to be covered by a single mutation. On the other hand, other branches without heavy arithmetic, where only few sensible mutation are applicable, would be valued lower due to the branch's lower amount of mutations.

Interchanging Relational Operators: The mutation of relational operators is also sensible but not in every case. It is not necessarily sensible to mutate `A \== B` to `A > B` because multiple cases exist where the semantics do not change, that is, every case where `A > B` is true. Moreover, we would need to know the types of the arguments in order to implement this mutation since arithmetic operators are only defined for integers, whereas non-equality is defined for all Prolog terms as well. A sensible mutant is created by negating relational operators (e.g. `<` to `>=`). By negating a relational operator, the semantics is inverted: every case which was true before will be false now and vice versa.

Furthermore, we decided to independently check edge cases of relational operators. For instance, by mutating `A >= B` to `A == B`. We then expect at least one test case to fail. Otherwise, no test case uses any values where A and B are not equal, which is either a bug or uncovered behavior. If only negating the relational operator for this example, a single test case using the same values for A and B will always lead to a dead mutant although there is no test case using different values for A and B. We thus deem the mutations covering the edge cases of relational operators to be sensible.

Negating a Predicate In Prolog, the negation of a predicate is implemented using negation as failure [2]. For instance, the goal `\+(p,q)` succeeds iff `(p,q)` cannot be proven by resolution. In propositional logic, the goal could be simplified by applying De Morgan's laws resulting in a disjunction with negated literals. Hence, we decided to directly negate operators where applicable instead of using Prolog's negation. For instance, we change disjunctions to conjunctions or negate relational operators as described above.

The data flow of a predicate in Prolog is usually defined by passing data among predicates within a conjunction. Disjunctions can be used to provide different clauses for a predicate. Negating a predicate within a conjunction or disjunction likely manipulates a program's data flow due to the forced backtracking and possibly occurring side effects (see Figure 4). We thus negate predicates within conjunctions or disjunctions using Prolog's negation as failure. In this context, we do not negate predicates which we have considered independently like relational operators in case Prolog's negation has the same effect. For instance, mutating a predicate `A >= B` to `A < B` and `\+(A >= B)` would result in equivalent mutations. We restrict this mutation to predicates within conjunctions or disjunctions since negating the only goal within a predicate's body would be equivalent to removing the whole predicate in case the goal has no side effects.

```
rev([H|T],Rev) :-                 rev([H|T],Rev) :-
    rev(T,NT) ,                       \+ rev(T,NT) ,
    append(NT,[H],Rev).               append(NT,[H],Rev).
```

Fig. 4. An Example for Negating a Predicate Resulting in Different Semantics.

3.2 Foolish Mutations

There are also several mutations suggested by Toaldo and Vergilio that we consider to be foolish. Our reasoning is the following:

Clause Reversal: By reversing the order of clauses of a given predicate, semantically different mutants are most likely just an infinite loop in case a predicate is non-deterministic (see Figure 5). The creation of a non-terminating loop has no value for calculating the mutation score, because one cannot tell whether the mutated branch is tested or not due to the non-termination. Therefore, we consider the result of a test that exceeds a reasonable time limit and a failing test to be different. Otherwise, the mutation score would possibly be corrupted.

Furthermore, reversing the order of a predicate's clauses does not change the semantics in case the predicate is purely logical and deterministic, that is, each clause of the predicate validates its inputs (see Figure 6).

```
add_to_list(L, _, 0, L).          add_to_list(L, E, C, R) :-
add_to_list(L, E, C, R) :-            CC is C - 1,
    CC is C - 1,                      LL = [E|L],
    LL = [E|L],                       add_to_list(LL, E, CC, R).
    add_to_list(LL, E, CC, R).    add_to_list(L, _, 0, L).
```

Fig. 5. An Example for a Non-Terminating Loop with Reversed Clause Ordering.

```
is_list([]).                      is_list([_|T]) :-
is_list([_|T]) :-                     is_list(T).
    is_list(T).                   is_list([]).
```

Fig. 6. An Example for an Equivalent Program with Reversed Clause Ordering.

Cut Transformations: Inserting, removing and permuting cuts often does not cause a change in semantics. In Prolog, there are two kinds of cuts: A cut is called *red*, when its removal would create a semantically not equivalent program. Predicates with red cuts are, thus, not pure in a logical sense and, per definition, behave differently without their cuts. All other cuts are called *green*: while not affecting the semantics of the program, they are used in order to increase performance. In practice however, it is difficult to efficiently decide whether a cut is red or green.

In the context of mutation testing, we only want to mutate red cuts in order to change the semantics. Shifting a red cut to a subsequent position is unlikely to change the semantics since the condition of the cut has still been evaluated. Thus, shifting red cuts to a prior position within the predicate is most likely to be reasonable for mutation testing, because the original condition after which a cut is called has not been evaluated. By preventing backtracking, the semantics of the predicate is likely to be changed as can be seen in Figure 7.

```
filter(Pred,[H|T],[H|NT]) :-      filter(Pred,[H|T],[H|NT]) :- ! ,
    call(Pred,H) , ! ,                call(Pred,H) ,
    filter(Pred,T,NT).                filter(Pred,T,NT).
```

Fig. 7. An Example for Moving a Red Cut Resulting in Different Semantics.

4 A Mutation Testing Framework

In the following, we will take a closer look at the implementation of our framework. As it relies on term expansion, the framework must be loaded before the module under test. Before the tool is able to begin with the mutation testing process, some setup routines are executed: in order to generate sensible mutants, the tool collects the source code's predicates during term expansion. Moreover, the term expander adds a *dynamic* declaration for all predicates that may be modified, in order to be able to use retraction and assertion to create mutants. Then, the actual tests are executed on the original code to ensure that every test passes. Otherwise, the criterion that tests pass on the mutant is obviously flawed. Furthermore, the overall runtime of the tests is stored in order to derive a reasonable timeout. As mutations might result in infinite loops, tests must be executed with a timeout on mutants.

The process can be divided into the following procedures (also cf. Figure 8):

1. find a suitable mutant, i.e., a predicate where a new mutation is applicable
2. generate a mutated predicate
3. retract the predicate and assert the mutated one
4. run the tests and check for failing tests or timeouts
5. restore the original predicate

This workflow continues until no other suitable mutation can be found.

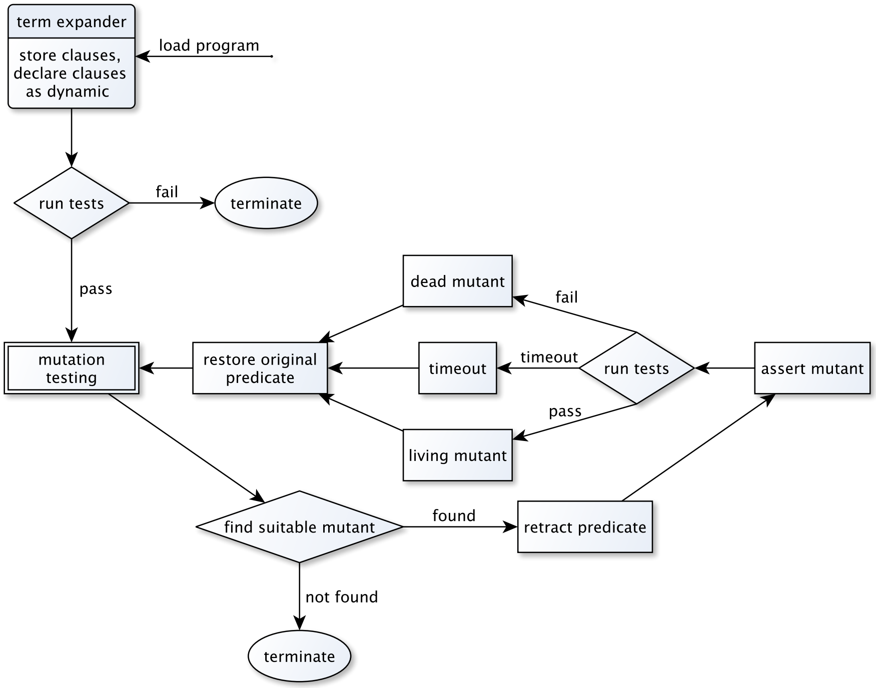

Fig. 8. The Framework's Workflow Diagram.

Particular care has to be taken when manipulating a single clause of a predicate: the order of all clauses should (usually) be retained, but new facts can only be asserted either at the top or bottom of the predicate. The tool thus retracts and re-asserts the entire predicate, i.e., all of its clauses, at once.

The mutation score is later calculated on the basis of all collected results. Mutants are labeled either as *dead* when at least a single test has failed or as *alive* when all tests have passed.

However, a mutation may have caused an infinite loop. To counter this, tests on a mutant are executed with a timeout which is defined by doubling the original runtime. Mutants which exceeded their test runtime are labeled as *timeouts*. In general, it is undecidable whether an actual infinite loop was encountered or if the mutant just runs significantly longer.

To calculate the mutation score, mutants labeled as timeouts are not considered. We thus deem the result of a test exceeding a time limit to be different from a failing test. Otherwise, the mutation score might be corrupted in case of detecting false positives caused by a significantly longer runtime.

5 Empirical Evaluation

In this section, we aim to evaluate two different aspects of mutation testing. Firstly, we verify our claims from Section 3 by measuring living and dead mutants on several pieces of code that we regard as reasonably tested. Secondly, the overall mutation scores for these programs are compared with the coverage computed by predicate, clause and sub-goal coverage.

For the evaluation, we use several Prolog programs which can be found on GitHub[3] along with a more detailed description. Most of the programs are part of an evaluation of different interpreter designs [11]. We have chosen these programs since they are part of a publication and have been developed test driven. Therefore, we expect these programs to have at least a mediocre test coverage. Additionally, we test the coverage of a translation between two formalisms (`alloy2b`) [9]. We think this program is interesting for a comparison of different coverage metrics since it only contains integration tests. The code will thus not be tested in detail which probably has an impact on the different coverage scores.

5.1 Sensibility of Mutations

A detailed overview of the results for our benchmarks can be found in Table 1, where for each considered file, the amount of living and dead mutants are given per mutation.

As claimed, removing a tested predicate always creates dead mutants. In cases where mutants are still alive, there simply existed no test case for the predicates. This is a mutation that is obviously sensible.

In our benchmarks, disjunctions have been fairly scarce; this is due to the fact that usually two separate clauses are preferred. However, no mutants on the few disjunctions survived the testing, so it seems to be as sensible as claimed. Changing conjunctions to disjunctions, however, generates more living mutants than expected. Apparently, this mutation is not as sensible as assumed.

There are a few instances of living mutants after negating a unification: e.g., in the case of the `fifteen_puzzle` data set, the only unification unifies a potential solution to the actual solution as a condition to terminate. Since the implemented algorithms are expected to always find a solution, terminating with a wrong solution after mutation is not caught. Overall, it still seems to be a sensible mutation for most applications.

[3] `https://github.com/hhu-stups/prolog-mutation-testing`

Table 1. Overview of Living (First Number) / Dead (Second Number) Mutants in Real-World Examples. Timeouts are Not Considered.

Mutation	ast_interpreter	compiler	parser	rational_trees	rt_bytecode	alloy2b	fifteen_puzzle
remove predicate	0/3	4/11	0/5	0/5	0/4	8/68	4/17
; to ,	0/2	0/2	0/3	0/2	0/1	1/15	1/0
, to ;	0/30	10/34	1/14	4/12	12/31	38/245	22/22
= to \=	0/1	0/2	0/0	0/1	0/0	4/51	1/0
\= to =	0/0	0/0	0/0	0/0	0/0	0/1	0/0
=:= to =\=	0/0	0/0	0/0	0/0	0/0	0/0	0/0
=\= to =:=	0/0	0/0	0/0	0/0	0/0	0/0	0/0
== to \==	0/2	0/0	0/0	0/3	0/1	1/2	0/0
\== to ==	0/0	0/0	0/0	0/0	0/0	0/0	0/0
> to =<	0/0	0/0	0/0	0/0	0/0	0/2	1/2
>= to <	0/0	0/4	0/7	0/0	0/0	0/0	0/0
< to >=	0/0	0/2	0/0	0/0	0/0	0/1	0/1
=< to >	0/0	0/0	2/5	0/0	0/0	0/0	0/1
+ to -	0/0	0/6	0/0	0/0	0/0	0/2	0/2
- to +	0/0	4/35	0/0	0/5	0/0	0/1	0/2
* to +	0/0	0/0	0/0	0/0	0/0	0/0	0/0
/ to -	0/0	0/0	0/0	0/0	0/0	0/0	0/0
> to ==	0/0	0/0	0/0	0/0	0/0	0/2	1/2
>= to ==	0/0	0/4	2/5	0/0	0/0	0/0	0/0
< to ==	0/0	0/2	0/0	0/0	0/0	0/1	0/1
=< to ==	0/0	0/0	2/5	0/0	0/0	0/0	0/1
increase number	0/0	10/34	11/3	0/0	0/15	9/151	2/30
decrease number	0/0	9/35	12/2	0/0	1/14	10/151	4/28
negate predicate	0/19	6/24	0/7	0/9	2/29	15/102	6/22
true to false	0/3	0/2	0/0	0/2	0/0	1/2	0/0
false to true	0/3	1/2	0/0	0/0	0/0	8/0	0/0
var to _	33/184	122/245	0/69	5/103	133/252	244/1327	95/138
atom to _	0/0	1/2	0/0	0/4	1/3	48/185	3/0
[] to _	0/1	12/6	3/0	0/1	2/1	42/51	11/1
permute cut	1/0	4/1	0/1	2/0	16/0	18/21	1/3
reverse predicate	3/0	13/2	3/2	5/0	4/0	38/22	18/0

For the considered programs, negating predicates within conjunctions and disjunctions results in a very small amount of living mutants. As this is fairly close to sub-goal coverage, we think that they might be created by uncovered code instead of false positives.

Changing variables to anonymous variables results in a fairly large amount of mutants in many cases, yet the mutation is applicable in a significantly larger amount of places as well. There is a good chance that tests do not cover all variables, in particular where entire predicates remain uncovered. Overall, this mutation seems to be sensible as well.

As expected, permuting cuts to an earlier position and reversing clauses of a predicate often results in semantically equivalent code.

Unfortunately, the selected examples do not use many arithmetic or relational operators. For most of these examples, however, there is no difference between negating a relational operator and using only the edge case. Only the mutation `>=` to `==` seems to be more sensible than using the negated operator for the parser example, that is `>=` to `<`. Here, two mutations stay alive indicating that there is indeed code that is not covered by any test.

Overall, our reasoning in Section 3 seems to be supported by our measurements. The only unexpected outcome is changing conjunctions to disjunctions. We think that specific tests are not as good as we initially expected and succeed for not equivalent mutations that cause variables being uninstantiated. For instance, if a unit test verifies the value of a variable using the Prolog unification (`=/2`) rather than equality (`==/2`), a predicate returning a variable will pass this test. When passing arguments between two predicates within a conjunction, the choice point introduced by a disjunction possibly causes uninstantiated variables. Another reason could be that arguments are not passed directly within a single but several conjunctions. Then, the mutation of a single conjunction does not necessarily change the semantics.

Yet, for example, mutations concerning arithmetic or relational operators are not covered by our benchmarks representatively. For these transformations, further code examples are required.

5.2 Comparison with Predicate and Clause Coverage

In the following, we compare the coverage of different Prolog modules using the coverage tools of SWI[4] and SICStus[5] Prolog as well as the introduced mutation testing framework. To give a short impression of the complexity of a program, we list the number of predicates, clauses and lines of code for each Prolog program in Table 2.

The coverage tool of SICStus Prolog measures how many times specific parts of the program, referred to as coverage sites, were executed. According to the documentation of SICStus, a coverage site corresponds to all predicate calls

[4] `http://www.swi-prolog.org/pldoc/man?section=cover`

[5] `https://sicstus.sics.se/sicstus/docs/4.3.2/html/sicstus/Coverage-Analysis.html`

like in *trace* mode. In consequence, there are different ways of interpreting the coverage results of the SICStus Prolog coverage analysis. First, we compute the coverage on the level of clauses, that is, we view a predicate's clause to be uncovered if any coverage site within this clause is indicated to be uncovered. Second, we compute the coverage on the level of predicates which we view as uncovered if they contain an uncovered clause. Third, we compute a more detailed sub-goal coverage where we view each sub-goal independently.

The coverage tool of SWI Prolog behaves similarly and computes the predicate coverage. In the following, we will thus only refer to clause and predicate coverage without distinguishing between SWI and SICStus Prolog.

Table 2. Comparison of Prolog Coverage Tools

Prolog File	LoC	Predicates	Clauses	Clause Coverage	Predicate Coverage	Sub-Goal Coverage	Mutation Coverage
ast_interpreter	107	2	20	100.00%	100.00%	100.00%	88.10%
compiler	165	15	42	80.95%	62.50%	86.04%	69.90%
parser	80	38	16	100.00%	100.00%	100.00%	92.20%
rational_trees	39	4	10	100.00%	100.00%	100.00%	96.50%
rt_bytecode	105	4	33	93.93%	50.00%	95.24%	70.10%
alloy2b	725	74	198	84.41%	75.95%	87.68%	80.20%
fifteen_puzzle	161	21	44	77.27%	67.10%	82.28%	70.40%

For two source files, we encountered technical issues with our mutation testing framework. Both programs rely on writing and consulting Prolog files at runtime. Yet, when mutating the code, the corresponding streams might not be closed properly, resulting in an error caused by holding too many file handles simultaneously.

Overall, the results are non-binary. In most cases, the results are similar to the clause coverage approach. For some files, our framework reports a higher score than predicate coverage. Yet, it is able to find uncovered instances where all other approaches claim perfect coverage. In general, no approach can fully substitute the others. Thus, we recommend to use mutation testing as an additional tool.

A rather unsatisfying result, however, is that for now living mutants require manual review in order to determine whether they are semantically equivalent to the original code. While this problem is generally undecidable, techniques from *declarative debugging*[6] [15] could be used to offer tool support as follows.

The idea behind declarative debugging in Prolog is to examine the proof tree [14] computed during execution. Each node inside the tree represents an atomic goal that succeeded. Calls are nested, i.e., the children of a node are the successful subgoals taken from the body of the predicate used to prove said goal.

Declarative debugging now classifies these nodes as either being *correct* (i.e., the goal is valid and all instances are true with respect to the intention of the

[6] Initially named *algorithmic debugging* by Shapiro [21]

programmer) or *erroneous* (i.e., not valid). Using this information, a declarative debugger can identify bugs, i.e., instances which are not supposed to be true in the intended interpretation, yet they are. Furthermore, it can use the proof tree to trace the location of programming errors.

In the context of mutation testing, we know several goals to be *correct*, as they have been traversed by the initial, successful, test run. Furthermore, we know where mutations took place and which calls should not be *errorneous*. Hence, the results of mutation testing can be used to provide input to a declarative debugger which could now be used to identify the root cause of a mutation being alive and help to distinguish between foolish mutations and missing test cases.

6 Related Work

As already stated in Section 3, different mutation operators for Prolog have been outlined by Toaldo and Vergilio [23]. We have evaluated each reasonable mutation independently using several examples in order to classify them as either sensible or foolish in Section 3. Our evaluation performed in Section 5.1 has shown that not all of these mutations are efficient, for instance, they might generate numerous semantically equivalent mutants. We have compared our implementation with several different coverage metrics in Section 5.2. Moreover, we have incorporated new mutations and made our implementation publicly available both for SICStus and SWI Prolog.

Of course, mutation testing can be performed on other, non-logical, languages as well. Among the most prominent tools is PIT [3], a mutation testing tool for Java. Imperative languages aside, mutation testing has been considered for declarative and functional languages as well [13]. Usable tools exist, for instance, for Haskell [12].

Since generating and testing redundant mutants makes up the majority of the runtime of mutation testing, plenty of work, e.g., by Offutt et al. [17], has been done on finding a set of mutants that is as small as possible but still finds all uncovered code. Ammann et al. [1] compare two different sets of mutation operations for PIT. We are not aware that comparisons of the amount of generated alive and dead mutants per mutation rule has been done for other languages.

This might be because in "traditional" languages, e.g., Java, C or even Assembly, control flow is usually linear, rendering selection of mutation operations fairly straightforward. In Prolog however, it is easier to manipulate control flow, e.g., changing conjunctions to disjunctions basically introduces if-then-else constructs. Futhermore, for many possible mutations, it is hard to predict how the program will be influenced, e.g., moving cuts forwards or backwards. An empirical study as shown in this paper renders it easier to reason about these transformations which simply are not possible in other programming languages.

Regarding test coverage, several measurement tools are available and integrated into the most common Prolog interpreters such as SWI [24] and SICStus [22]. While they provide basic code coverage metrics, they usually only report on reached ports, that is, call, exit, redo and fail during execution. As discussed in the introduction and shown in our empirical evaluation, this can lead to different results compared to calculating coverage based on mutation testing. Of course, this does not imply that one metric is better than the other but rather that a combination of multiple approaches yields the best results.

7 Future Work

Even though we have improved the selection of mutation operators, our approach still generates mutations semantically equivalent to the program under test. As an equivalent mutation does not lead to a failing test by definition, each equivalent mutation causes a false positive to be reported. With our current implementation, deciding whether a mutation is semantically equivalent is done after the test results are reported. In particular, the decision is made manually by the programmer as mentioned at the end of Section 5.2.

To improve the efficiency of our test framework, techniques to detect semantically equivalent mutations should be incorporated in the future. For instance, we are able to approximate the check for equivalence of two programs by restricting the domains of the arguments and using constraint solving to search for a counter example. However, Prolog is a dynamic language not providing types which hampers constraint solving. To counter this, we could try to detect the types of a predicate's arguments at runtime. Unfortunately, this is not possible in general, for instance, if an argument is a variable which is not unified within a specific predicate call. *plspec* [10], for example, offers a simple and easily extensible domain specific language for type annotations. If a predicate is annotated using *plspec*, we are able to derive the types of arguments as well as their role, that is, whether they act as an input or output of a predicate. One downside of approximating the equivalence of two programs is that we do not consider possible side-effects of a predicate. Nevertheless, in practice, we expect an approximation with appropriate domains to exclude more false positives than true positives. The meaningfulness of mutation testing will thus probably increase.

Another line of future work is the integration of automated test case generation in the mutation testing framework. Again, we need to be able to derive the types of the arguments of a predicate (e.g., assume that *plspec* has been used). We gain a lot of information about a program under test during mutation testing. In case a mutation is alive, we might be able to use the mutated and the original code to generate an appropriate test case covering the mutated clause. For instance, we could mutate a predicate p using two arguments while the first argument acts as an input and the second one as an output. The predicate is mutated to p_{mut} and the mutation stays alive. Assuming the mutation did change the semantics of p, we know that our test case needs to satisfy a call to p and has to fail for p_{mut}.

We can then use techniques such as fuzzing to generate randomized inputs until a set of parameters has been found. Furthermore, we can use constraint solving to search for appropriate arguments. Doing so, we can, on the one hand, search for any arguments that cause different behavior on the level of the predicate call.

That means, the arguments satisfy the constraint $\exists a, b : p(a, b) \wedge \neg p_{mut}(a, b)$. On the other hand, in case we know which arguments are inputs and outputs, we can probably generate a more detailed test case by searching for input values that cause different output values. In the context of our example that results to asserting $\exists a : p(a, b) \wedge p_{mut}(a, \bar{b}) \wedge b \neq \bar{b}$ to hold. Of course, the implementation of the predicate p might be faulty. We thus cannot automatically determine a generated test case to be correct. However, we can present generated test cases to the user who can validate the behavior.

Additionally, as discussed in Section 5, not all mutations were covered by our benchmarks. Therefore, the impact of these mutations should be measured on other Prolog programs.

8 Conclusion

We have presented a framework for performing mutation testing on Prolog code. Starting from the discussion of mutation rules by Toaldo and Vergilio [23], we have devised a set of mutation rules we deem sensible, i.e., we suspect them to mostly compute semantically different mutations. Our testing framework is available both for SICStus and SWI Prolog and can be downloaded from

`https://github.com/hhu-stups/prolog-mutation-testing`.

We have shown empirically, that our mutation testing framework can handle different Prolog programs. In particular, we tested it on large examples, showing both applicability and performance of our approach. Furthermore, we have shown that mutation testing indeed reports different coverage statistics than the ones provided by the coverage analysis tools shipping with SICStus and SWI. We do not want to start a discussion on whether predicate coverage, path coverage or MC/DC analysis is better or worse than mutation testing. Instead, we argue that any further knowledge about test coverage and the validity of a test suite helps to improve the overall implementation.

References

1. P. Ammann, M. E. Delamaro, J. Offutt, et al. Establishing Theoretical Minimal Sets of Mutants. In *IEEE International Conference on Software Testing, Verification, and Validation, 7*. IEEE Computer Society, 2014.
2. K. L. Clark. Negation as Failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977.*, pages 293–322, 1977.
3. H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A Practical Mutation Testing Tool for Java (Demo). In *International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 449–452. ACM, 2016.
4. A. Denisov and S. Pankevich. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, April 2018.
5. A. V. Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring Test Code. In *International Conference on Extreme Programming and Flexible Processes in Software Engineering*, pages 92–95, 2001.

6. B. J. Grün, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *IEEE International Workshop on Mutation Analysis*, pages 192–199. IEEE, 2009.
7. Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
8. S. Krings. Code Coverage Analysis for Prolog. Bachelor's thesis, Heinrich-Heine-University, Duesseldorf, Germany, 2 2010.
9. S. Krings, J. Schmidt, C. Brings, M. Frappier, and M. Leuschel. A Translation from Alloy to B. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - International Conference, ABZ*, pages 71–86, 2018.
10. P. Körner and S. Krings. plspec - A Specification Language for Prolog Data. In D. Seipel, M. Hanus, and S. Abreu, editors, *Declare 2017*, volume 499 of *Technical Report*. University of Würzburg, 2017.
11. P. Körner, D. Schneider, and M. Leuschel. Evaluating Interpreter Design in Prolog. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung KPS*, Schriftenreihe des Instituts für Computersprachen, 2015.
12. D. Le, M. Alipour, R. Gopinath, and A. Groce. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *International Symposium on Software Testing and Analysis, ISSTA*, 2014.
13. D. Le, M. A. Alipour, R. Gopinath, and A. Groce. Mutation Testing of Functional Programming Languages. Technical Report. Oregon State University, School of Software Engineering and Computer Science, 2014.
14. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, 1984.
15. L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997.
16. A. J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.
17. A. J. Offutt, G. Rothermel, and C. Zapf. An Experimental Evaluation of Selective Mutation. In *International Conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.
18. C. H. Papadimitriou. A Note the Expressive Power of Prolog. *Bulletin of the EATCS*, 26(21-23):61, 1985.
19. R. S. Sangwan and P. A. L. LaPlante. Test-Driven Development in Large Projects. *IT Professional*, 8(5):25–29, Sept. 2006.
20. D. Schuler and A. Zeller. Javalanche: Efficient Mutation Testing for Java. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 297–298. ACM, 2009.
21. E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.
22. SICS, Kista, Sweden. *SICStus Prolog User's Manual.* Available at `http://www.sics.se/isl/sicstuswww/site/documentation.html`.
23. J. R. Toaldo and S. R. Vergilio. Applying Mutation Testing in Prolog Programs. In *VII Workshop de Testes e Tolerância a Falhas*. Biblioteca Digital Brasileira de Computação, 2006.
24. J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.