Bachelor Thesis

A Tool for Mutation Testing of Prolog Exercise Tasks

Ivan Khu Tanujaya

September 2021 – December 2021

Supervisor:

Prof. Dr. Janis Voigtländer
M.Sc. Oliver Westphal

Open-Minded

Universität Duisburg-Essen
Fakultät für Ingenieurwissenschaften
Abteilung Informatik und angewandte Kognitionswissenschaft
Fachgebiet Formale Methoden der Informatik
47048 Duisburg

Abstract

Automation is a huge help to conduct E-Learning. Autotool is a tool that can support that statement, and this tool helps in grading students' exercise tasks. However, with Autotool, adequate test cases are required so that educators can be sure that the submitted solutions are acceptable. To help educators obtain adequate test cases for Prolog exercise tasks, this thesis applies mutation testing that is normally used to test large projects in software development. This uncommon application of mutation testing is supported with a proposed tool created with ReactJS Framework in JavaScript programming language. The tool will generate mutants of different types and show their test results. With this result, the test cases of an exercise task could be improved. If an educator wants to introduce a new exercise task, a strategy is required to design test cases efficiently. This thesis proposes and discusses a method for that purpose. Moreover, this thesis evaluates the test suites of sample exercise tasks available. Leak findings, which is an occasion where a mutant generated by the tool can pass through all the test suites are recorded and further explained.

Contents

Chapter 1

Introduction

The current growth in the technology sector brought multiple effects into various sectors in the world. One of the effects of this growth involves the education sector and elevating education to the next level. With this, new learning methods such as E-Learning has been introduced and is commonly used by many. E-Learning makes teaching both easier for educators as well as students. The most seen proof of it is the flexibility E-Learning offers. There is no expectation for students to be present in a room during a specific time window but only to study the materials uploaded by the educators anytime within the given deadline.

An aspect of E-Learning is the utilization of automation in deploying exercise tasks. Imagine a situation where 2000 students are enrolled in a course with a set of exercise tasks weekly. Achieving this without automation would mean a considerable amount of resources (or "educator") is required to make this work, for both creating the tasks and grading them. With automation, the effort or resources towards effective teaching would be less than it would be without. An example of an automated exercise tool is the Autotool[Aut], which is also currently used by the chair of Formal Methods in Computer Science at the University of Duisburg-Essen.

The chair of Formal Methods in Computer Science at the University of Duisburg-Essen teaches multiple lectures. One of them is the main focus in this thesis which is the "Programming Paradigm" lecture. This lecture teaches the functional programming language "Haskell" and logic programming language "Prolog" with the help of Autotool[SVW19]. The primary audience of this lecture is students who are assumed to have a beginner-intermediate level of programming skills. Another lecture named "Introduction to Logic" also teaches Prolog, where the primary audience is students with a beginner level of programming skills. Hence, this proposed tool serves the purpose of supporting the idea of providing effective teaching for the group of students, regardless of their level of programming skill at the moment. In a way, this tool will help the process of creating Prolog exercise tasks and help to assess the coverage of the test suites of Prolog exercise tasks.

The process of creating Prolog exercises has its own difficulties. One of them is not having enough test coverage to check the correctness of codes submitted by students. This proposed tool which is developed has the capability of covering those limitations. The tool will perform mutation testing, which is a type of software testing, to the exercise tasks available. This way, one can be sure that each exercise task has adequate test cases to prevent incorrect submissions. Considering that the scale of the exercise program is relatively small, one can deduce that performing mutation testing on these exercise tasks can be considered as unit testing. Unit testing in software development tests the minor components of a large project to make sure that each component functions appropriately[Alm14][Ham21b].

The mutation testing method involves the creation of mutants by the tool, which will be similar to the code generally submitted by students, which contains small mistakes. The reason behind this is that beginner Prolog programmers often make mistakes due to misunderstanding some syntax or operators and ignoring several specific rules in Prolog, as Prolog is quite a unique language compared to C, Java, and others. The tool would then generate these mutants and test whether each mutant is accepted as a correct program or rejected as a wrong program. With this result, the educator could induce whether or not additional test cases are required to be added into the test suite. This way, this educator could run the solution through this program to ensure enough test coverage on each code and prevent incorrect submissions from being accepted.

The mutation types implemented in this tool extend to the work done by Toaldo, Vergilio[TV06], and Efremidis et al.[Efr+18]. This paper deal with a more specific kind of mutant or incorrect codes, which students typically submit in an exercise task. Also, the number of killed mutants is not considered to be important, which means that the score of each mutation type is not taken into account. The workflow in this tool utilizes the help of services offered by a server written in Haskell. These services include Prolog interpreter, parser, and task-checker.

In the next section of this paper, the three entities related to this paper such as the Prolog programming language, Autotool, and mutation testing, will be discussed further. Next, the mutation operator types performable by this tool will be explained, which is an extension of the related works[TV06][Efr+18]. After that, the implementation methods of the proposed tool, as well as its features, will be put into discussion. In the end, the analysis of the tool regarding the results will be shown, and the whole paper will be summarized.

Chapter 2

Fundamentals

2.1 Prolog

Prolog is a logic programming language lectured at the university by the chair of Formal Methods in Computer Science[SVW19]. The author believes that understanding the following terminology would be sufficient to understand the whole thesis. The following explanation is based on the author's experience with Prolog as well as the author's understanding of the Prolog slides, which are used in the lecture and can be found on the following GitHub page[1]. Additionally, a valuable overview of Prolog notions can also be viewed on page 25 of the slides.

2.1.1 Prolog Objects

Term

A fundamental entity in Prolog is a term. A term in Prolog could be a constant, a variable, or a structure. In a constant, all numbers and atoms belong to this group. An atom is an identifier that is usually written in a camelCase format. Meanwhile, a variable is an identifier that is usually written in a PascalCase format. In some cases, the underscore character "_" might also be found in an atom as well as in a variable. Last, a structure is a composition of terms and usually starts with a functor followed by its parameters wrapped within a round bracket. Each of the functors, as well as the parameters, are also considered as a term.

It is possible that a functor exists inside the structure, making that functor with its parameters a substructure to that structure. To be more precise, consider a structure A with a functor x and its parameters. In one of the parameters of functor x, there is another functor y with its parameters. This functor y and its parameters make up a new structure B, a substructure of structure A.

[1] https://fmidue.github.io/ProPa-Slides/2021/Prolog.pdf

For example:

```
shape(circle,rgb(50,150,X))
```

In the example above, shape, circle, and rgb are examples of an atom as well as a constant. 50 and 150 are numbers, which also belong in constants. X is a parameter of the functor "rgb" and is a variable. In the provided example, there are two functors, which are "shape" and "rgb". The functor "shape" with its parameters make up structure A which contains another functor "rgb". This functor "rgb" with its parameters make up another structure B, a substructure of structure A. As an alternative, shape/2 and rgb/3 are also valid ways of representing the functors "shape" and "rgb". The number at the end represents the arity, which is the number of parameters of each functor.

A special type of term in Prolog is a ground term, defined as a structure or a term without any variable. The following modification will transform the previous example of the "shape" structure into a ground term:

```
shape(circle,rgb(50,150,250))
```

List

In Prolog, a list is a unique structure. It is usually represented by several terms inside a square bracket. There are numerous ways of representing a list. In Prolog, there is a built-in operator, which is the character (|). This character separates the head and tail elements within a list. Note that this tail element itself is a list. Another way to build a list is with the help of a dot (.) functor. This functor will take two parameters, the first one being a term and the second a list object or an empty list.
For example, ".(1, .(2,[ ]))" will build up into "[1, 2]".

For a better understanding, the list representations below are equivalent in Prolog:

```
[1, 2, 3, a]
[1| [2, 3, a]]
[1, 2| [3, a]]
.(1, .(2, .(3, .(a, [ ]))))
[1, 2| .(3, .(a, [ ]))]
[1, 2, 3| .(a, [ ])]
```

Operators

Operators are the entities that normally perform some functionality. In other words, they are also functors from symbols or special characters. These operators have precedence or order of priority on which operator to be executed initially if multiple operators exist in an expression without separation. In this thesis, mutations are performed on operators such as relational operators, arithmetical operators, and others. There is an alternative way to write these operators that is to write them infix. The arithmetical expression $1 + 2 * 3$ can also be re-written as $+(1, *(2, 3))$. Note that due to * having higher precedence than +, Prolog will execute this operator first. More information regarding precedence can be accessed in the documentation[2].

2.1.2 Prolog Syntax

Moving on to the more practical terminology of Prolog, on top of the hierarchy is the clause. A clause can either be a fact, a rule, or a query and always ends with a full stop (.). The simplest of the three is the fact. A fact comprises a predicate with several parameters, which are always a constant or an anonymous variable. A single underscore character (_) represents an anonymous variable in Prolog. Together, a predicate with its parameters wrapped within a round bracket will make a literal. Apart from being a constant or an anonymous variable, the parameter of a predicate in a literal could also be a standard variable. These literals would primarily be used in defining rules.
A rule is somewhat more complicated than a fact. It is made up of a left-hand side and a right-hand side literal separated by an implication(:-).

A collection of defined facts and rules will make up a database used to run Prolog queries. Typically a Prolog query would return a Boolean type result True or False, or some constants in which the query satisfies the database. Below is an example of Prolog facts, rules as well as queries:

```
landTransport(car).
landTransport(bus).
seaTransport(ship).
seaTransport(boat).
airTransport(plane).

transportation(X) :- landTransport(X) ; seaTransport(X) ; airTransport(X).
```

[2] https://www.swi-prolog.org/pldoc/man?section=operators

With this database of facts and a rule above, running the following queries below will give the results of:

```
?− landTransport(car).
true.

?− seaTransport(car).
false .
```

Notice that the literal *"landTransport(car)."* is contained in the database but not *"seaTransport(car)."* Hence the result is as shown above.
Another type of query will return the constants that satisfy the query:

```
?− airTransport(X).
X = plane.

?− seaTransport(X).
X = ship;
X = boat;
false .

?− transportation(X).
X = car;
X = bus;
X = ship;
X = boat;
X = plane.
```

The first two queries are relatively easy, as Prolog will look into the database for the constants that satisfy the given query. However, in the second query, Prolog will return the result *"X = boat."* and look after the next one from the database. As Prolog does not find another result, it will also return *"false."* as a response. On the contrary, when Prolog found *"X = plane"* in the first query, it is the last fact in the database and satisfies the query. As a result, Prolog stops looking for the next answer and does not return *"false."* at all.

Our database defined a rule for the predicate transportation with the help of conjunction (;) operators. In English, the rule says that X is a transportation if X is either a landTransport, a seaTransport, or an airTransport. Hence Prolog will look for all X that satisfy the query *"?-landTransport(X)."* and return those. Next, it will look for all X that satisfy the query *"?-seaTransport(X)."* and lastly, the query *"?-airTransport(X)."*. Because the fact *"airTransport(plane)."* is at the end of the database, Prolog returns *"X = plane."* as the last result.

2.2 Autotool

Autotool is an E-Learning platform that utilizes automation for exercises in Theoretical Computer Science. This program[Aut] was written in 1999 to help students execute derivation in Hilbert Calculus(for propositional logic) and has been optimized since then to be used for various other exercises. Today, the platform can automatically check exercises for Prolog programs. The Autotool is a free software released under the GNU / GPL License. This Autotool has a database and a web front-end to help manage organizational instances such as semesters, lectures, students, results, and high scores.

In general, Autotool has two different interfaces for educators and students. The students have only access to see the exercise tasks, task descriptions, and a solution textbox for each task. On the other hand, the educator must create the task configuration where the task description and the test cases are contained. These test cases would then be used to check the correctness of the program code submitted by the students. Therefore, this collection of test cases is one of the main stars of the whole process in this automated exercise grading program. The components in the configuration are divided into multiple sections. A section can either be a comment block or a regular code block. A comment block starts with the characters (/*) and ends with (*/). Additionally, each line in a comment block starts with the character(*). The first comment block in the configuration will always be the test suite, and all the other blocks after it are the task description. The presence of three or more dashes (- - -) in the task configuration indicates that the upcoming blocks will be hidden from the students. Another aspect of designing a test suite is that the educator must remember that the test cases should terminate before the timeout limit, as there is a timeout for each test case. Any test which does not terminate before this limit will be considered a fail test.

When a student submits a program code, the program will be tested with test cases in the form of queries. After submitting, the two possible outputs are “Success, ok” or “Failure, No...”. In case of a failed test, the program will stop testing the submitted code with the following test cases, and the test case behind the failure will be shown. In other words, the submitted code did not comply with a test case in the configuration and failed. There is a fascinating implementation of the Autotool test case, which is the existence of hidden tests. These tests only differ by an escape character at the beginning of the line, but it significantly affects the whole testing and teaching process.

In the case of failing a regular test case, the Autotool feedback will show which test case or query was unsuccessful and hence consider the program as incorrect. With this feedback, a student can bypass this failure by adding the missing literal so that the test case or query would be successful in the upcoming tests. This, however, is not

a proper way of correcting submission. Hence, the program will fall into the trap of the hidden test where Autotool will only return feedback that the program code has failed some test, without specifying which. This way, one could be sure that an effective teaching and learning process is conducted.

Another feature that is available in Autotool is the derivation trees. Unfortunately, this feature is not integrated yet with the proposed tool. However, the derivation tree is a massive assist for an educator to design a test suite. In the case of a successful test of a mutant, the derivation tree provides a valuable insight into the reason behind the mutant passing the test. Suppose a mutant stayed alive after a test. With the help of the derivation tree, the educator could incrementally add new test cases into the collection that would kill the mutant.

2.3 Mutation Testing

Mutation testing is one type of software testing which is also known as a fault-based testing. Toaldo and Vergilio quoted, that according to Mathur and Wong[MW94] as well as Wong[Won93], some empirical studies show that fault-based criteria or the so-called mutation testing is the most efficacious to reveal faults[TV06]. As the name suggests, this type of software testing involves the creation of multiple mutants. A code with several modifications will create a mutant of that code. In other words, a mutant differs syntactically and semantically from the original code on a tiny scale. Changes to the code can be made by changing operand in an expression such as replacing "x >= y" into "5 >= y" or changing the operator in the expression such as "x >= y" into "x < y" or even expression modification such as "x==5" into "NOT x==5"[Ham21a]. A mutant that cannot pass through a collection of test cases is called to be dead. Oppositely, a mutant that passes the test cases is called an alive mutant. The mutation score is recorded at the end of the test to examine how adequate the existing test cases are. The desired score is 100% which means that no mutant passes the test. Hence the test cases are adequate. To calculate the mutation score, the following formula will be used:

$$Mutation\ Score = (Dead\ Mutants/No.\ of\ Created\ Mutants) * 100 \tag{2.1}$$

Such mutations that alter the code semantic are considered sensible and those that kept the code semantic are considered the foolish ones[Efr+18]. In chapter 4, both the foolish and sensible mutation types are listed out and further explained. With the help of mutation testing, one can increase the test suite's quality to kill any mutant. This quality improvement is achievable as mutation testing is a type of white box testing where it is possible to check the changes made to the mutant code at the end of each test. Hence, an extra test case could be added in order to kill the mutant.

There are already tools for mutation testing for programming languages other than Prolog. To the best of the author's knowledge, an existing tool for Prolog programming language is only the MutProlog tool[TV06] and a framework made by colleagues from Heinrich Heine University in Düsseldorf[Efr+18]. However, when applying mutation testing into Prolog programs, a non-standard mutation is required. In other words, there is some mutation which only makes sense and applies to Prolog programs. For example, mutating the parameter used to define a function would make no point if it is done to a function in C program. A function "int f(int x, int y)" mutated into "int f(int x, int x)" would not make any sense if it is done in C program, as the compiler will be strict enough to return an error while compiling. On the other side, mutating a Prolog function definition "f(X, Y)" into "f(X, X)" would still be compiled smoothly in Prolog.

In this context, as Prolog language is being taught to beginner programmers, one cannot deny the chance of these tiny mistakes existing in codes submitted by the students. Hence is this proposed tool responsible for replicating these mutants with tiny mistakes and eliminating the probability of these incorrect codes being accepted in the submission.

In this thesis, the mutation score is not focused on as the primary goal. Instead, the purpose of the proposed tool is to make sure that the test cases are adequate or the coverage of the test suite is comprehensive enough. Hence, if a mutant passes the test, an action could be done by investigating the mutant and deciding which test case should be added to kill the passing mutant. Note that in a normal situation, mutation testing is performed on projects with a larger objective instead of an exercise task of a smaller objective. Hence, the aim is towards finding alive mutants in order to be able to kill them by adding test cases and eventually eliminating the chance of Autotool accepting a faulty submission.

To prevent false submissions from students, another important aspect of writing a test suite in Prolog is to have test cases that are expected to be true and test cases in which the program should fail. In other words, suppose there is an example predicate "testPredicate(...)", then there should be a test case where it would return true and another one which it would return false. In the tool, this is applicable by having a test case "testPredicate(...)" as well as "not(testPredicate(...))". Obtaining a true as a result of the "not(testPredicate(...))" test case means that it is false.

Chapter 3

Related works

There are papers regarding mutation testing in general. However, only the works on [Efr+18] and [TV06] are the closest related to this thesis as they both apply mutation testing in the Prolog language. Also, as mentioned before in the first chapter, this thesis works on the basis of what was done previously on these papers.

The work of Toaldo and Vergilio[TV06] laid down mutation operators applicable for Prolog programs, and these mutation operators are used in their proposed tool named "MutProlog." In the latter work done by Efremidis et al.[Efr+18], mutation operators outlined by Toaldo and Vergilio[TV06] were evaluated and each one of them was classified as to whether the operator was sensible or foolish. A foolish mutation is defined as the mutation which retains the code's original semantic. On the other hand, a sensible mutation is a mutation that alters the code's original semantic. The mutation operators which are performable by the proposed tool are implemented based on the evaluation done by Efremidis et al.[Efr+18]. Only the sensible mutations mentioned are performable by this proposed tool. However, there is a difference in the target or the scale of the work done on this thesis. The mutation testing in this thesis focuses mainly on exercise tasks and not on programs in real-world software development.

More regarding the mutation operators implemented by the proposed tool will be explained in chapter 4. If the mutation operators implemented by Efremidis et al.[Efr+18] were written in C language and Linux operating system. The mutation operators implemented in this paper were written in JavaScript within Windows operating system or in Haskell, if the mutation operators are hosted externally on the server.

Chapter 4

Mutation Operators

In general, the tool offers two modes in which a mutation can be performed. They are the individual mode and summary mode.

In individual mode, only one change per mutant will be performed. For this mode, the user is also required to input the number of mutants they wish to obtain from the mutation. If the desired amount is greater than the maximum amount of mutants possibly created, the tool will still give out the maximum amount possible. Hence it is safe to input 1000 in order to obtain all the mutants in individual mode. The maximum amount of mutants a user can obtain from this mode is equal to the total number of operators contained in the code, on which the transformation can be performed.

In summary mode, the tool will look at all the possible mutation variations that are performable on the code with the help of a function similar to the replicateM function in Haskell. That means there is a mutant with only one transformation and another mutant that has all the operators transformed in a mutation type. In this mode, the user's input on the 'number of mutants' field will determine the maximum amount of mutant (variations) to be created. Again, similar to individual mode, if the user's input is greater than the number of possible variations, the tool will only give out all possible variations. In general, the amount of possible variations is defined as such:

In a mutation where,

$$x = Number\ of\ possible\ variations\ in\ one\ position$$

$$and$$

$$y = Number\ of\ positions\ where\ transformations\ can\ be\ performed,$$

the maximum number of possible mutants would be $x^y - 1$. The reduction by one is due to unchanged code being considered one mutant variation. However, such code is

identical to the original code and hence should be exempted. Our tool will not produce this kind of code that resembles the original code.

Additionally, it is advised to keep the user's input number on 1000 at the maximum for performance aspect. If the amount of possible variation is greater than the user's input, then the selection process will begin. This process is as such, that a random combinations of the numbers 1 and 0 will be produced in an array, with each array having the amount of numbers equal to the amount of operator positions (or changes performable on the code) in a mutation type. The number 1 will indicate that the operator should be transformed, and 0 indicates otherwise. This process will be repeated until the number of mutants in the result equals the user's desired amount, without any duplicate.

4.1 Implemented Mutation Operators

The following mutation operators are offered by the proposed tool. These mutation operators are inspired by the previous works[TV06][Efr+18]. Hence, these mutation operators might contain a few variations and not replicate what is done in the previous works. Moreover, the tool offers extensibility which means that some mutation operators are implemented and hosted on the server and not in the tool. Such mutation operators are called external mutation operators, and they consist of "Clause Removal" and "Variable to Anonymous Variable". Note that the implementation of these two external mutation operators differs from the other mutation operators. The differences are specified in the description of each mutation operator. The mutation operators implemented are:

4.1.1 Conjunction to Disjunction

In this mutation, the tool will change a conjunction operator into a disjunction operator within a clause. Changing the logic between disjunction to conjunction will alter the semantic of the code. Performing this mutation in **individual mode** will flip one conjunction operator per mutant. On the other side, performing this mutation in **summary mode** will create a number of mutants, with each mutant having a different amount or different positions of conjunction operator flipped.
For example, consider the following clauses within a Code:

```
uncle(X,Y) :− brother(X,Z), child(Y,Z).
grandson(X,Y) :− male(X), child(X,Z), child(Z,Y).
```

(i) An example of a mutant that will be produced when a user performs the mutation in **individual mode** would be:

```
uncle(X,Y) :− brother(X,Z); child(Y,Z).
grandson(X,Y) :− male(X), child(X,Z), child(Z,Y).
```

In this example, there are three conjunction operators in the code, which means that there are three possible mutants to be created by performing the mutation in this mode. Each mutant has only one conjunction operator transformed into a disjunction operator. Notice that by flipping the conjunction operator in the uncle rule, the semantic of the code has changed. In the mutant, all child predicates will be considered as uncle as well.

(ii) An example of a mutant that a user would obtain when the mutation is performed in **summary mode** is:

```
uncle(X,Y) :− brother(X,Z); child(Y,Z).
grandson(X,Y) :− male(X), child(X,Z); child(Z,Y).
```

In this case, performing the mutation in **summary mode** will flip more than one conjunction operator. In this example, the first and the third operators are flipped out of the three operators, leaving the second one unchanged. However, another mutant exist, where the second and the third operators are flipped with the first operator unchanged. Following the rules above, the amount of possible mutants created here is $2^3 - 1 = 7$. Also, notice that this mode will alter not only the semantic of one predicate but multiple predicates.

4.1.2 Disjunction to Conjunction

On the other hand, changing a disjunction operator into a conjunction operator will also alter the semantic of the code. The individual mode of this mutation type works similarly to subsection 4.1.1, where only one disjunction operator will be flipped per mutant. Likewise, performing this mutation on the summary mode will give the user a number of mutants. Each mutant has a different amount or variation of disjunction operators being transformed into conjunction operators.
Consider the following clauses as an example:

```
parent(X) :− father(X,_); mother(X,_).
child (X) :− father(_,X); mother (_,X).
```

(i) An example of a mutant that will be produced when the tool performs the mutation in **individual mode** would be:

```
parent(X) :− father(X,_), mother(X,_).
child(X) :− father(_,X); mother (_,X).
```

The tool will be able to detect the two disjunction operators within the clauses. In one mutant, an operator will be flipped, and in another, the other operator will be flipped. In the provided example, the disjunction operator in the parent(X) rule is flipped, and now the predicate has a different semantic than the original, which is that a parent should be a father and at the same time a mother.

(ii) An example of a mutant which the user would obtain when this mutation type is performed in **summary mode** is:

```
parent(X) :− father(X,_); mother(X,_).
child(X) :− father(_,X); mother (_,X).
```

Following the same rules, in this example, there would be three possible mutants to be created at most if the tool performs this mutation type in summary mode.

4.1.3 Interchanging Relational Operators

This mutation type will detect all the relational operators in the code and transform them, depending on the mode. However, this tool will restrict the transformation, which means that not every operator will be transformed to all the other operators. For example, this tool only considers the transformation of (=) into (\=) and not (=) into (=<).

The tool will flip one arithmetical operator per mutant in individual mode, complying with the transformation rule. In summary mode, all the possible variations of changes will be given out, but each operator will be transformed according to the rule. Below is a list of transformations that will be performed by the tool:

- = to \=
- \= to =
- =:= to =\=
- =\= to =:=
- == to \==
- \== to ==
- > to =<
- >= to <
- < to >=
- =< to >

Given the following clauses as an example:

```
fromTo(N,M,L) :- N > M, L = [ ].
fromTo(N,M,[N|L]) :- N =< M, N1 is N+1, fromTo(N1,M,L).
```

(i) An example of a mutant that will be produced when the tool performs the mutation in **individual mode** would be:

```
fromTo(N,M,L) :- N > M, L = [ ].
fromTo(N,M,[N|L]) :- N > M, N1 is N+1, fromTo(N1,M,L).
```

The tool will detect the three relational operators in the code. Notice that in the provided example, the last relational operator (=<) is transformed into (>), according to the rule defined previously. In this mode, there are three mutants at maximum. Transforming relational operator changes for sure changes the semantic of the predicate.

(ii) In summary mode, the user would obtain this mutant as an example:

```
fromTo(N,M,L) :- N =< M, L = [ ].
fromTo(N,M,[N|L]) :- N > M, N1 is N+1, fromTo(N1,M,L).
```

In the provided example, two out of the three operators are flipped according to the rule. Following the calculation formula, there are $2^3 - 1$ possible mutants to be generated.

4.1.4 Interchanging Arithmetic Operators

This mutation works similarly to the one in subsection 4.1.3. It will detect all the arithmetical operators in the code and transform some of them, depending on the mutation mode. Also, the transformations should comply with the transformation rule of arithmetical operators. In individual mode, only one arithmetical operator per mutant will be flipped. In summary mode, the tool will give out all the possible variations of mutants. Below are the transformations which will be performed by the tool:

- + to -
- * to +
- - to +
- / to -

An example code for this mutation type:

```
test(A,B,C,D,E,F,G,H,I) :−
V1 = ((H∗10000) + (D∗1000) + (G∗100) + (A∗10) + (B∗1)),
V2 = ((I∗100) + (H∗10) + (A∗1)),
Sum = V1+V2,
Sum =:= (C∗10000) + (F∗1000) + (F∗100) + (F∗10) + (E∗1).
```

(i) An example of a mutant that will be produced when the tool performs the mutation in **individual mode** would be:

```
test(A,B,C,D,E,F,G,H,I) :−
V1 = ((H∗10000) + (D∗1000) + (G∗100) + (A∗10) + (B∗1)),
V2 = ((I∗100) + (H∗10) + (A∗1)),
Sum = V1−V2,
Sum =:= (C∗10000) + (F∗1000) + (F∗100) + (F∗10) + (E∗1).
```

There are in total 24 arithmetical operators in the example, and hence there are also 24 possible mutants to be produced by this mutation type in the individual mode. One example mutant shown above has an addition operator in the variable Sum(line 4) being flipped into a subtraction operator.

(ii) In summary mode, the user would obtain this mutant as an example:

```
test(A,B,C,D,E,F,G,H,I) :−
V1 = ((H∗10000) + (D∗1000) + (G∗100) + (A∗10) + (B∗1)),
V2 = ((I+100) + (H+10) + (A+1)),
Sum = V1−V2,
Sum =:= (C∗10000) + (F∗1000) + (F∗100) + (F∗10) + (E∗1).
```

Notice that all the multiplication operators within the predicate V2 are flipped into addition operators in this mutant variation. Coincidentally in the provided example, the previous change performed on the individual mode(line 4) is also performed on this example. The example code contains 24 arithmetical operator and two alternating variations per operator. This means that there is a total of $2^{24}-1$ possible mutants, which is greater than an advised number of 1000 mutants. Hence, selecting the right number of desired mutants matters in this case.

4.1.5 Predicate Negation

In this mutation type, the mutation will only take place to structure predicates and not to expressions. This mutation operator will either negate a predicate or remove an existing negation. Negating a predicate will most likely change the semantic of the code and hence is considered a non-foolish mutation. Consider the following clause:

```
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).
```

(i) An example of a mutant that will be produced when the tool performs the mutation in **individual mode** would be:

```
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).
```

In the provided example, there are three structure predicates. Hence for this mutation type in individual mode, the tool can generate a maximum of three mutants.

(ii) In summary mode, the user would obtain this mutant as an example:

```
\+ 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).
```

Notice that in the example mutant, the tool negates the predicate solve and removes the negation on the predicate test. In summary mode, there are $2^3 - 1$ possible mutants.

4.1.6 Clause Removal

In this mutation type, the tool will detect all the clauses in the code and remove one clause. This mutation operator is one of the external mutation operators available in the tool and is implemented differently from the others. This implementation aims to check whether it is possible to outsource several mutation operators externally to the server. In the tool, this mutation operator has another name, which is “Drop Clause Mutation”. On the day this thesis is submitted, the implementation of this mutation operator is in a way described below.

Performing this mutation in any mode will only perform one change per mutant, which means that the maximum number of possible mutants is equal to the number of clauses in the code. Also, in this mutation operator, the entry for "number of mutants" is only relevant to individual mode. Performing this mutation in summary mode will return all the possible mutants to the user, with each mutant having one clause removed. Consider the following example:

```
mother(X,Y) :− female(X), child(Y,X).
brother(X,Y) :− male(X), child(X,Z), child(Y,Z), X\=Y.
uncle(X,Y) :− brother(X,Z), child(Y,Z).
grandson(X,Y) :− male(X), child(X,Z), child(Z,Y).
```

An example mutant would be:

```
mother(X,Y) :− female(X), child(Y,X).
uncle(X,Y) :− brother(X,Z), child(Y,Z).
grandson(X,Y) :− male(X), child(X,Z), child(Z,Y).
```

In this example, there are four clauses. This means that in the current implementation, performing this mutation in summary mode will return the maximum number of possible mutants, which means four mutants will be generated. The individual mode is useful to generate a desired number of mutants, while the summary mode will generate all possible mutants.

4.1.7 Variable to Anonymous Variable

This is another mutation operator which is implemented and hosted on the server. In the current implementation, this mutation type functions a bit different than the other internal mutation operators. This mutation operator will detect all the variables within a clause and replace one of them into an anonymous variable. Depending on the mode, the tool will transform all the variable of a name, on both left and right hand side of the clause defining that predicate. On the day this thesis is submitted, this mutation operator is implemented differently than the way it is described below. The differences are explained at the end.

For this mutation type the tool do not consider the amount of transformable operator/position to be a deciding factor, but the number of variable names declared within the clauses. We consider a variable within a clause to be a change. So a clause with three unique variables declared will be considered as three changes. Following this logic, a code with three clauses, with each clause having three unique variables, will mean that there are nine possible changes to the code.

In individual mode, each mutant will have exactly one variable to be transformed into an anonymous one. Using the previous example (three clauses with three variables each), this means that there are nine possible mutants in this mode for this mutation type.

In summary mode, the user will get a number of mutants with each mutant having different amount or variation of changes to the atoms or variables. Following the logic, in the previous example there would be $2^9 - 1$ mutants possible for this mode.

Consider the following concrete code example:

```
fromTo(N,M,L) :- N > M, L = [ ].
fromTo(N,M,[N|L]) :- N =< M, N1 is N+1, fromTo(N1,M,L).
```

(i) Performing this mutation in **individual mode** will return the following example mutant:

```
fromTo(N,M,L) :- N > M, L = [ ].
fromTo(N,_,[N|L]) :- N =< _, N1 is N+1, fromTo(N1,_,L).
```

and

```
fromTo(N,M,L) :- N > M, L = [ ].
fromTo(N,M,[N|L]) :- N =< M, _ is N+1, fromTo(_,M,L).
```

The tool will detect three unique variables on the first clause and four unique variables on the second clause. On the second clause these unique variables consist of N, M, L, and N1. This give a total of seven variables the tool could perform the mutation on. Hence, there are seven possible mutants to be created in individual mode. In other words, there are seven possible changes to the code.
Notice that on the second example, the variable N1 which does not appear on the left hand side of the clause would still get noticed by the tool and transformed into anonymous variable.

(ii) Performing this mutation in **summary mode** will return the following example mutant:

```
fromTo(N,_,L) :- N > _, L = [ ].
fromTo(_,M,[_|L]) :- _ =< M, N1 is _+1, fromTo(N1,M,L).
```

With seven possible changes, following up the previous logic would give the user $2^7 - 1$ possible mutants.

On the day this thesis is submitted, the current implementation of this mutation operator differs in some way. The individual mode for this mutation operator is not implemented at all. In summary mode, the mutant number is not implemented yet, which means that performing this mutation will return all the possible mutants without cap. Hence, the user has to be careful with performing this mutation operator due to computational power.

4.2 Other Mutation Operators

The mutation operators below were not implemented in the tool due to various reasons. The mutation operator "Clause Reversal" and "Cut Transformations" were exempted as they were considered foolish[Efr+18]. Hence, in this paper, there is only a brief explanation with an example for the two foolish mutation operators. Moreover, the concept of cut transformations is not introduced in the lecture. The mutation operator "Interchanging List Syntax" was not assessed in the previous paper. However, this mutation is considered a "student-specific" mutation operator. Therefore the concept of this mutation operator is described for future reference.

4.2.1 Interchanging List Syntax

This mutation type will modify the representation of a list. One typical misunderstanding that often happens to students is to understand the difference between the operators (,) and (|) in a list. So for this mutation operator, the tool will detect all the (,) and (|) in the code and perform the mutation to one another. Consider the following code:

```
g(_,[ ],[ ]).
g(_,[Y],[Y]).
g(X,[Y,Z|Zs],[Y,X|Us]) :- g(X,[Z|Zs],Us).
```

In the example, the tool will detect five list operators, which are contained in the third clause. This amount translates to $2^5 - 1$ mutant possibilities in total, each varying in the number of changes. Consider the following mutant example:

```
g(_,[ ],[ ]).
g(_,[Y],[Y]).
g(X,[Y|Z|Zs],[Y,X|Us]) :- g(X,[Z,Zs],Us).
```

All the (,) and (|) operators within a list should be detected by the tool. The mutant example has two changes performed on the third clause. Moreover, the mutant example has five list operators, which translates to $2^5 - 1$ possible mutants.

4.2.2 Clause Reversal

If a predicate is defined multiple times by the code, a Mutant could be created by reversing the order of the definition. For example, given the following predicate definition:

```
connection(X, Y) :- direct(X, Y)
connection(X, Y) :- connection(X, Z), direct (Z, Y)
```

After applying the mutation, the predicate will be defined as:

```
connection(X, Y) :- connection(X, Z), direct (Z, Y)
connection(X, Y) :- direct(X, Y)
```

If the predicate is defined in more than two clauses, then a random reversal might happen. It is also possible to reverse the clause depending on the user's demands.

4.2.3 Cut Transformations

This mutation will involve rules containing cuts. The program will shift the cut forward and change the semantic of the predicate. For example, given a predicate **max** to determine the higher value among the two Variables:

```
max(X,Y,Z) :- X =< Y, !, Z=Y.
max(X,_,X).
```

Applying this mutation will return an example mutant:

```
max(X,Y,Z) :- !, X =< Y, Z=Y.
max(X,_,X).
```

Chapter 5

Implementation

The tool which is developed is a single-page web application integrated into a server that will run locally. The front-end is mainly written with JavaScript and the help of several libraries to render the components such as React and Bootstrap. However, the server used is a Scotty server developed by the chair of Formal Methods in Computer Science and written with Haskell.

Since the tool is a single-page web application, the client-server architectural pattern is applied. Establishing the connection between the web application and the backend server was a massive priority of this thesis and a core element of the tool. This connection lays the foundation for upcoming future works such as new mutation operators, which the chair of Formal Methods in Computer Science could implement with any programming language.

During the development, various kinds of designs are considered to be implemented. The goal or the priority of developing this tool is to keep everything as efficient and straightforward as possible. However, there are some difficulties with styling due to multiple components changing the native component styling. Hence, extra work was to be done, which was to separate the styling within two similar components. An example is the result table component, which shows the test result of mutants in a table, collides with the DiffView, which is made up of a table as well.

There is also a Continuous Integration used in the whole development process. After every pushed commit or whenever there is a pull request, it is made sure that the whole code is clean of warnings or errors. In other words, it is to be sure that the whole code can be compiled and built without error or warnings. Moreover, in the whole development process, reusing codes are prioritized to maintain the tool's modularity.

The whole development process of this program is divided into four main tasks:

1. Development of the web application front-end
2. Establishing connection with the server and its functionality
3. Rewriting Prolog solution code from a structure
4. Implementing mutation operator functions

5.1 Tasks

5.1.1 Development of the web application front-end

The idea of having a front-end for the tool was to promote easy access for the whole mutation process. The author believes that having a front-end for the tool is more user-friendly than having a user run multiple command lines in the PowerShell. As the web application is a single page, every functionality in the tool will be easily reachable by the user once the user has uploaded the two essential files. These are the configuration file and the solution file.

The front-end of the program is developed with the help of ReactJS and some integrated styling libraries such as Bootstrap. The current single-page web application can be pulled from the fmidue repository[1]. In this repository, the Scotty server is also contained. The development of the front-end was conducted gradually. In other words, the first implementation of the front-end provides very basic functionality. This functionality allows the user to upload two files required for the whole process, which are the configuration file and the solution file. The configuration file of the Prolog task contains the test cases and the task description, while the solution file contains the code on which the mutation operators are going to be performed. This basic implementation is now the tool's main page, shown initially every time a user runs the tool. On the day this thesis was submitted, there were extra functionalities such as:

1. Feature to edit configuration, solution, and mutant file
2. Feature to retest the file after editing
3. Feature to see which part of the solution file has been mutated

[1] https://github.com/fmidue/ba-ivan-khu-tanujaya

5.1.2 Establishing connection with the server and its functionality

The server is contained and can be pulled from the same repository as the web application. This server is required as the web application uses functions such as Prolog task checker, Prolog file parser, and other functions which are hosted on this server. The task checker is responsible for returning the result whether the solution code passes the test suite contained in the configuration file. However, the writing of this server is not a part of this thesis as it was developed by the chair of Formal Methods in Computer Science.

5.1.3 Rewriting the code from a structure

One of the difficulties of implementing the mutation functions with JavaScript was to identify whitespace within characters. For example, consider the following two expressions in the form of strings:

```
A > B
A>B
```

Using normal JavaScript syntax manipulation to perform mutation on the relational operator(>) would not be trouble-free because of the existence of whitespace. This issue is problematic in this and many other cases whenever an inconsistency in whitespace exists. Another alternative was to use RegEx to find the target characters to be transformed. However, this is also not an effective method generally. In the case of the solution code having a nested structure such as parentheses inside a parenthesis, the usage of RegEx will fall apart. As a solution, this tool utilizes the "parse file" function from the server. The input to execute this function from the server is a prolog code file. In return, the server will send back the structure of the prolog code, which is similar to a nested object in JavaScript. From this structure, the tool will then rewrite the actual code with every operator's position recorded. This makes performing mutations to the code easier. With the help of the position data, JavaScript syntax manipulation will be effective as the operators are changed based on their positions.

There are, of course, other alternatives, such as performing mutations directly on the structure provided and un-parse the structure back into raw code. However, it was decided to implement the function that rewrites the code from the structure. Nonetheless, there is a limitation that is realized for now. If a new mutation operator is introduced, there might be the need to modify this unparsing function. The difficulty of this modification depends on the type of the mutation operator itself.

5.1.4 Implementing Mutation Functions

In principle, there are seven mutation operators able to be performed by the tool. These mutations were explained in the catalogue in chapter 4. These functions are written with JavaScript and utilize JavaScript syntax manipulation.

An interesting aspect of the implementation is the centralized control by a single file called the registry. This registry controls every mutation operator shown in the tool. In this file, important information such as the default option and the mutation function location is contained. The registry also controls whether a mutation operator is enabled, that is if the tool will be able to perform the mutation operator. The mutation option helps to select the mode and desired number of mutants as well as which mutation operator to perform. For mutation operators, there is a default option fixed. This default option makes sure that a user can always press on "Start Mutation" button without having to modify the selection initially.

As mentioned previously, flexibility and extensibility are prioritized in the implementation of mutation operators. It was meant that not only the mutation operators in the tool are executable. Calling mutation operators which are hosted externally on a server is also possible. For example, the mutation operator "Variable to Anonymous Variable" is hosted on the Haskell server and is performable by the tool.

5.2 Usage

5.2.1 Tool walkthrough

When a user starts the web application and the backend server, they will land on the main page. On the main page, there are two file uploaders for the configuration file and the solution file of a corresponding Prolog exercise task. For the purpose of reassuring the user that the solution is correct, the tool will run an initial test, checking whether the solution code passes the test cases currently contained inside the configuration file. Then the result will be shown, whether it is "Success ok" or "Failure, No...". Below, these components will follow: the configuration editor, solution editor, and the mutation options performable to the solution code.

At this stage, the user can make changes to any of the editors, save the content temporarily, and re-test the combination. Once the user is satisfied, the user can proceed to the following procedure, which is to select the mutation options to be performed and input the amount of mutant desired. After some time, depending on the options selected, a result table will show up, showing which kind of mutants are created as well as the result whether they pass the test (stay alive) or fail the test (dead mutants). The

results in the table can be filtered so the user can select which mutation operator they wish to see with which test result. The entry will expand by clicking on the table entry, showing more details regarding the mutant, which is called a mutant card. At the very top of the mutant card, the difference between the mutant code and the original solution code will be shown. On the bottom of the mutant card is the area for the user to edit the mutant code and re-test it with the current content of the task configuration.

At this point, it is also possible to modify the configuration by editing the configuration editor, which is shown after the initial testing. By saving changes made to the configuration content and re-testing the mutant code, the mutant code will be tested with the new configuration content. The idea behind this is that only one configuration content/test suite is being used to test each mutant every time changes are made to kill a new mutant. The user can proceed from this point to incrementally add new test cases to kill the next mutants.

5.2.2 Proposed method

Suppose a user wants to find test cases for an exercise task, with only a few test cases currently contained in the configuration file. A proposed method of improving the test cases would be to perform all mutations available and analyze each mutant that stayed alive until the end of the process. By observing the code difference between the alive mutant and the solution code, the user could then add a test suite that would kill the alive mutant. After re-testing, the mutant will help to determine if the added test case is sufficient to kill the mutant. Note that as mentioned in section 2.3, in Prolog, it is also advised to have test cases that define what the code should do and test cases that define what the code should not do. By executing this sequence of observing and adding test suites to the configuration file iteratively until most of the mutants are dead, one can be assured that the test cases contained in the modified configuration file are optimal.

5.3 Graphical User Interface Overview

This web application is made up of various React components. However, there are component groups that are core parts of the web application and hence worth the spotlight. Here are the overviews:

1. **File Editor**
   The following view will appear once the user has uploaded the two essential files. The content of the editor fields can be modified and saved temporarily for the purpose of testing.

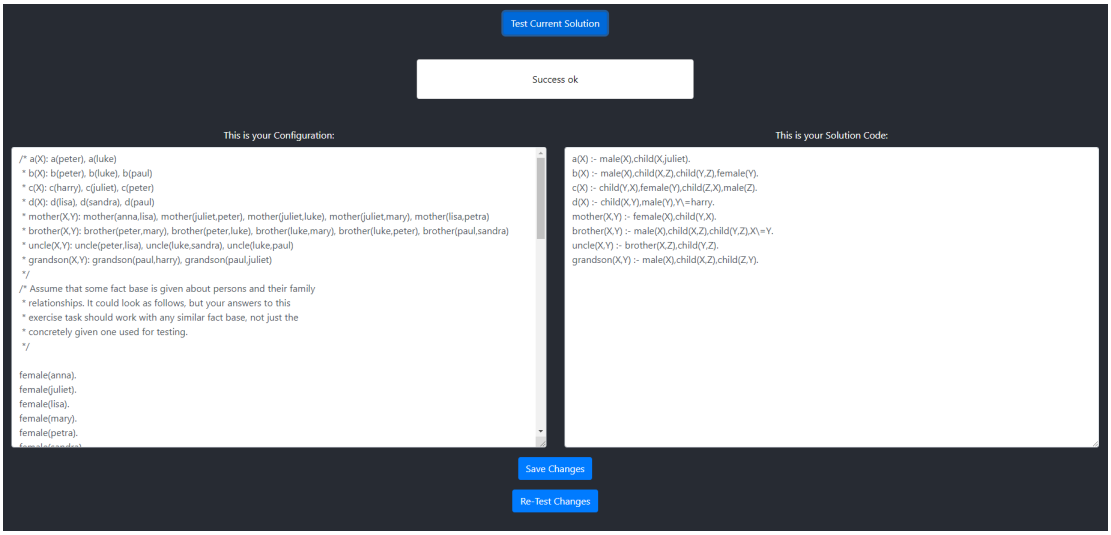

Figure 5.1: Main Editor Fields

2. **Mutant Card**

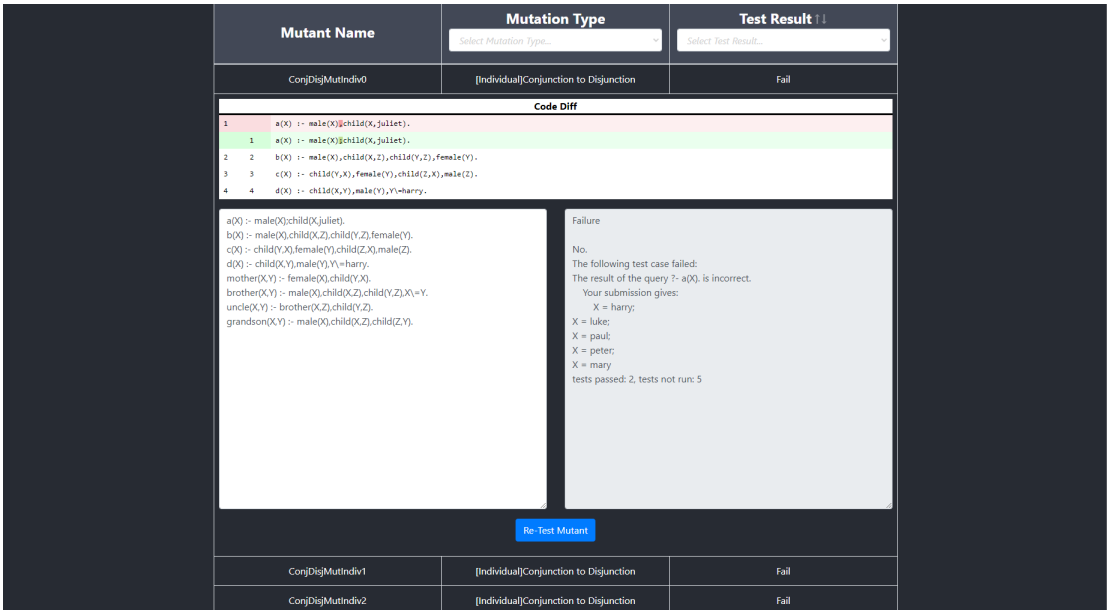

Figure 5.2: Mutant Card

Clicking on an entry of the result table will show the mutant card. This card will show the differences between the mutant code and the original code. Other than that, the raw code of the mutant and the test feedback will be shown. The re-testing button will test the mutant with the current configuration editor content, which is the editor on the left side shown in Figure 5.1.

Chapter 6

Evaluation

In this chapter, the tool is utilized, and some findings regarding the tool will be made. The method mentioned previously in section 5.2 will be put into practice and discussed. In this thesis, twelve pairs of configuration and solution files are provided by the chair of Formal Methods in Computer Science to be evaluated. One type of exercise task requires students to define Prolog predicates which are expected to do specific objectives. An example task of this type would be defining a predicate that checks whether an element is a penultimate element in a list. Another type of exercise task would be to formulate Prolog queries to be executed on a database of facts and rules.

6.1 Implementation of the method towards adequate test cases

In the exercise task where students are required to formulate Prolog queries for a Prolog database, this database might be provided, or the students might be required to create it with given predicate names. For this type of exercise, it does not require many expenses to come up with the minimal test cases, which means that for each predicate, there is a test case in which the whole predicates in the sample solution will pass. A method to reach a minimal test suite would be to put the inquired predicates on the test cases and set it to true so that when running it with the tester, it will show which results are possibly obtained when running the query from the solution into the database. From the shown result, add each literal/solution into the test cases of each predicate. Doing this iteratively until each predicate has a test case will result in minimal test cases, allowing the sample solution to pass.

From this point onward then the tool will come into play. The mutation operators will be performed on the solution case, which will generate a number of mutants and the result when the mutants are tested with the minimal test cases obtained from the previous step. After the first mutation run of the author's experiment, none of the mutants created stayed alive when tested with minimal test cases. Hence, it is concluded that for such exercise tasks where queries are to be formulated to be tested

on a database, reaching adequate test cases is quite uncomplicated.

When applying the method to the other task type that requires defining predicates, this process gets more complicated. We start with only having one test suite for this method, which is what the predicate is supposed to do, as described in the task description. For example, if the task is to define a penultimate predicate where "penultimate([a,b,c,d],c)" should hold, this exact literal will be the first test suite. After generating all the possible mutants for the sample solution and testing it with the test suite, all mutants which stayed alive until the end of the process will go into further observation. To focus on all the alive mutants, select "OK" at the table header of the result table. Then after clicking and expanding an entry, the user can observe the difference between the original code and the mutant provided. After observation, the user should introduce a new test case that would kill the mutant. Usually, this would be a test case that describes what the predicate is not supposed to do.

This sequence of observation and incrementally adding up a test suite would then be repeated to achieve optimal test cases. In the author's experiment, repeating this method to each of the tasks collection available, five iterations were required at the maximum for a task to obtain an optimal test suite. At this point, all the generated mutants are killed. One of the difficulties found to reach a state of an optimal test suite for such an exercise task is, for example, to find the effective test suite to rule out trivial predicate. For example, the tool will produce a mutant with a fact as "predicate(_,_,_)." which generally would return true or pass all the test cases for the predicate. However, there are also other interesting findings when trying the tool, which will be discussed in the next sections.

6.2 Findings during experimentation of the tool

When trying out the tool to generate optimal test suites for each of the exercise tasks in the collection, the author made the following findings towards the tool and the workflow, which is considered worthy of mention. These are:

Evaluation on the test suites acquired from the method

For each exercise task, it is iterated until an optimal collection of test suites is obtained. The number of iterations required per task varied. However, for some exercise tasks, having one positive and one negative test for each predicate in the exercise task would be sufficient to kill all the generated mutants. These optimal test cases are yet very minute compared to the test suites currently being used by the chair. In other words, the test cases currently in hand are more massive and cover many more grounds in

capturing incorrect student submissions. This proves that there is a need for a bunch of new mutation operators to be introduced to generate mutants of other kinds.

Effectivity of the mutation operators introduced

From all seven mutation operators implemented and performed on each of the solutions of the exercise tasks, it is observed which type is effective and which one is not. On the bright side, all mutation operators prove to have a purpose in helping to reach optimal test cases when the proposed method in subsection 5.2.2 is put into practice. None of the mutation operators implemented only produces dead mutants when tested with the initial test suite. However, it is noticed that an improvement towards the mutation operator "Predicate Negation Mutation" in subsection 4.1.5 is possible. In the current implementation, predicates on the left-hand side of a rule will also be mutated. This implementation is somewhat superfluous as the test result for these mutants will indicate syntax error and fails. Hence, an improvement towards this mutation operator would be to perform only to the right-hand side of a rule.

On the other side, the mutation operator "Variable to Anonymous Variable" proves to be very effective. The mutants generated by this mutation operator are helpful to deduce which test suites are required to kill mutants or trivial programs such as f(_,_,_), which would always pass all the positive tests for the predicate f.

Limitations and Improvements

While using the tool, a few limitations are identified. These limitations involve various aspects of the tool, from the components to the utility functions. The un-parser function that rewrites the raw code from a structure is one of these functions. One limitation is the lack of integration between the mutant card and the configuration editor field. Indeed it is possible to modify the content of the editor fields. However, the user is also required to collapse the mutant card and re-expand it to be assured that the mutant card has loaded the updated configuration editor content. There are also a few improvements found around this matter. First, to have an extra button and functionality to test all the currently generated mutant with the current content of the configuration editor and update the result accordingly to each entry. This improvement, in a way, will promote efficiency and help users put the proposed method into practice. The next improvement is to be able to save the current content of configuration or solution editor fields into a file in the device storage. Being able to save the current content will help the educator store the modified or improved test suites easily.

The DiffView component that shows the difference between the code in the solution editor field and the mutant proves to be very helpful in identifying which test

suites are to be added into the collection to kill the alive mutants. Nonetheless, this component is far from being perfectly helpful, as it will also show the difference in whitespace. This is due to the fact that when the un-parser function of JavaScript rewrites the code from the structure, all the whitespaces will be eliminated from the code. Hence, all mutants created by the tool will have no whitespace within characters. On the other side, the external mutation operators which are hosted on the server will not eliminate any whitespace from the existing code. As the solution editor field contains the code from the tool's un-parser function, there are many whitespace differences when compared to the mutant generated by an external mutation operator. At times, this might be disturbing as the user has to observe the difference precisely.

Another improvement idea proposed to help the user's analysis process is to implement the component and functionality to show the derivation tree so that the user who operates the tool could understand why an expected dead mutant stayed alive and passed all the existing tests. For the derivation tree to work, there is a precondition that the device has GraphViz[GN00] installed.

6.3 Assessment of the current test suites

From all of the exercise tasks put into trial, three exercise tasks are found to have leaks. In other words, for each of the three exercise tasks, there is at least a mutant that passes through all the current test suites used by the chair of Formal Methods in Computer Science to test submission codes. The reason varies, which will be discussed further.

Task A

Below is shown the solution of the first task as well as the alive mutant:

Solution Code:

```
f([ ], Ys,Ys).
f([X|Xs],Ys,[X|Zs]) :- f(Ys,Xs,Zs).
```

Mutant Code:

```
f([ ], Ys,Ys).
f([_|Xs],Ys,[_|Zs]) :- f(Ys,Xs,Zs).
```

The reason why this mutant passes through all the test cases is due to the variable X appearing only on the left-hand side of the clause. However, variable X is not a singleton variable. A singleton variable is a variable which only appears once in

a clause. Changing a singleton variable into any variable would create an alternative solution. In this case, when the variable X is substituted by an anonymous variable, it slightly alters the semantic of the code, and hence a specialized test suite is required to kill this mutant. This mutant able to leak through the current test cases proves that there is not enough coverage for this example task. The following test case was tested with the tool and proved to be able to eliminate this mutant:

not(f([1,2],[3,4,5],[5,5,5,5,5]))

Task B

In the other case, several mutants were able to leak through and pass the current test cases. Consider the following solution code:

```
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).

generate(A,B,C,D,E,F,G,H,I) :- permutation ([0,1,2,3,4,5,6,7,8,9],[ A,B,C,D,E,F,G,H,I,_]).

test(A,B,C,D,E,F,G,H,I) :- V1 = ((H * 10000) + (D * 1000) + (G * 100) + (A * 10) + (B
    * 1)), V2 = ((I * 100) + (H * 10) + (A * 1)), Sum = V1+V2, Sum =:= (C * 10000) +
    (F * 1000) + (F * 100) + (F * 10) + (E * 1).
```

Mutant Group 1

A huge portion of the alive mutants have changes done on the first clause and no other changes on the rest of the code from the solution code. The changes on the first clause can be one of the following:

```
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).

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).

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).

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).
```

After further inspection, these mutants could pass through due to a small mistake in the task configuration. In the task configuration, the definition of predicate solve is also written, and it is written outside any comment block. This made this predicate solve a valid code inside the program and would be used by the tester. In other words, changes made to the predicate solve in the solution code would not affect the testing as the definition of predicate solve in the task configuration could still be used.

A proposed solution to solve this issue is to put the definition of the predicate solve in the task configuration inside a comment block or remove the clause solve from the solution.

Mutant Group 2

Three last mutants from the alive mutants have their arithmetical operator transformed on the third clause. The mutants will look like any of these:

```
test(A,B,C,D,E,F,G,H,I) :- V1 = ((H * 10000) + (D * 1000) + (G * 100) + (A * 10) + (B
    * 1)), V2 = ((I * 100) + (H * 10) + (A * 1)), Sum = V1+V2, Sum =:= (C * 10000) +
    (F * 1000) + (F * 100) - (F * 10) + (E * 1).

test(A,B,C,D,E,F,G,H,I) :- V1 = ((H * 10000) + (D * 1000) + (G * 100) + (A * 10) + (B
    * 1)), V2 = ((I * 100) + (H * 10) + (A * 1)), Sum = V1+V2, Sum =:= (C * 10000) +
    (F * 1000) - (F * 100) - (F * 10) + (E * 1).

test(A,B,C,D,E,F,G,H,I) :- V1 = ((H * 10000) + (D * 1000) + (G * 100) + (A * 10) + (B
    * 1)), V2 = ((I * 100) + (H * 10) + (A * 1)), Sum = V1+V2, Sum =:= (C * 10000) -
    (F * 1000) - (F * 100) - (F * 10) + (E * 1).
```

Note that the (+) operators from each of the clauses are transformed into (-) and still can pass the test suites. The reason behind this is the task setup. All the solution variations for this setup involve the value 0 being assigned to the variable F. Thereupon, it does not matter if it is an addition or subtraction of the multiplication of variable F, or in another word, zero.
It is concluded that these mutants are alternative solutions in the given setup, where the value of variable F will always be zero.

Task C

In this last case, only one mutant was able to pass through. The code and alive mutant are shown below:

Solution Code:

```
flatten(T,L) :- flatten(T,[],L).
flatten(leaf(X),L,[X|L]).
flatten(node(T1,X,T2),L,L2) :- flatten(T2,L,L1),flatten(T1,[X|L1],L2).
```

Mutant Code:

```
flatten (T, L) :- flatten (T, [], L).
flatten ( leaf (X), L, [X|L]).
flatten (node(T1, _, T2), L, L2) :- flatten (T2, L, L1), flatten (T1, [_|L1], L2).
```

This mutant has the variable X of the third clause transformed into an anonymous variable. Again, the variable X is not a singleton variable. This indicates that the mutant is not an alternative solution. Indeed, the mutant is semantically different from the solution code. The mutant code will ignore the second parameter of the predicate node, which is not the case in the solution code. However, this task has a task description or an initial test case which says that "flatten(node(node(leaf(1),2,leaf(3)),4,leaf(5)),[1,2,3,4,5])" should hold.
Hence, by studying the initial test case, one can induce an additional test case to kill this mutant. This test case which was tested with the tool and proved to be able to eliminate this mutant, is:

not(flatten(node(node(leaf(1),2,leaf(3)),4,leaf(5)),[1,2,3,2,5]))

Chapter 7

Summary and conclusion

Mutation Testing is one method to make sure the correctness of a program code. This method is usually used to check massive programs and projects to ensure that every part of the program functions as desired. This is also applicable to the Prolog language, although it is pretty uncommon. However, there are a few papers regarding mutation in Prolog that can be found and are used as a basis of the mutation operators performed by the tool proposed in this thesis. It is to be highlighted that the purpose of this tool and thesis is to perform mutation testing on exercise tasks in a lecture, which is a tiny portion of real-life software development. The mutation operators could be conducted on various entities of a Prolog program such as variable, arithmetical operator, or even a clause as a whole. In this paper, there are also modes and mutant numbers introduced for each mutation operator type. Mutation mode will determine how many changes are to be made to a mutant. The number will determine the number of mutants obtainable from one particular mutation operator type.

The proposed tool is constructed with the ReactJS framework in JavaScript language. This tool has many functionalities on the user level, such as testing a pair of task configuration with the corresponding solution and making changes to both dynamically. The core functionality of the tool is how mutants are generated with their results shown, whether they stayed alive at the end of a cycle. By investigating some cases of the user's choice, the user could investigate each of them to add up test suites into the collection in the task configuration.

This thesis also proposed a method to find an adequate collection of test cases when an educator introduces a new exercise task into the collection. A collection of sample exercises have also been evaluated with the proposed tool, where some findings regarding the tool's performance and the current sample exercises were made. These findings prove the insufficiency of the test suites currently used and at the same time, prove the contribution this tool can make. However, this tool is still far from perfect, as many changes can be introduced towards the improvement of the tool, or new mutation operators could be introduced to generate mutants with huge diversity. Nevertheless, with the current implementation, the tool proves to be an assist in terms of evaluating the current test suites and reaching adequate test cases for each exercise task.

Bibliography

[Alm14] Misja Alma. *Mutation Testing: How Good are your Unit Tests?* https://xebia.com/blog/mutation-testing-how-good-are-your-unit-tests/. Accessed on 18.11.2021. Nov. 2014.

[Aut] Autotool. https://gitlab.imn.htwk-leipzig.de/autotool/all0/-/blob/master/README.md. Accessed on 18.11.2021.

[Efr+18] Alexandros Efremidis et al. "Measuring Coverage of Prolog Programs Using Mutation Testing". In: *CoRR* abs/1808.07725 (2018). arXiv: 1808.07725. URL: http://arxiv.org/abs/1808.07725.

[GN00] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[Ham21a] Thomas Hamilton. *Mutation Testing in Software Testing: Mutant Score And Analysis Example*. https://www.guru99.com/mutation-testing.html. Accessed on 18.11.2021. Oct. 2021.

[Ham21b] Thomas Hamilton. *Unit Testing Tutorial: What is, Types, Tools And Test EXAMPLE*. https://www.guru99.com/unit-testing-guide.html. Accessed on 18.11.2021. Oct. 2021.

[MW94] Aditya P. Mathur and W. Eric Wong. "An empirical comparison of data flow and mutation-based test adequacy criteria". In: *Software Testing* 4 (1994).

[SVW19] Marcellus Siegburg, Janis Voigtländer, and Oliver Westphal. "Automatische Bewertung von Haskell-Programmieraufgaben". In: *Proceedings of the Fourth Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2019), Essen, Germany, October 8-9, 2019*. Ed. by Sven Strickroth, Michael Striewe, and Oliver Rod. Gesellschaft für Informatik e.V., 2019. DOI: 10.18420/abp2019-3.

[TV06] Juliano R. Toaldo and Silvia R. Vergilio. *Applying Mutation Testing in Prolog Programs*. 2006.

[Won93] W. Eric Wong. "On Mutation and Data Flow". PhD thesis. USA: Purdue University, 1993.

List of Figures

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere weiterhin, dass ich diese Arbeit noch keinem anderen Prüfungsgremium vorgelegt habe.

Duisburg, den 02. Dezember 2021

.....

Ivan Khu Tanujaya