

Tuomo Hopia

**MUTATION TESTING IN FUNCTIONAL
PROGRAMMING**

Master's Thesis in Information Technology

December 27, 2023

University of Jyväskylä

Faculty of Information Technology

Author: Tuomo Hopia

Contact information: `tuomo.hopia@gmail.com`

Supervisor: Jorma Kyppö and Antti Valmari

Title: MUTATION TESTING IN FUNCTIONAL PROGRAMMING

Työn nimi: MUTAATIOTESTAUS FUNKTIONAALISESSA OHJELMOINNISSA

Project: Master's Thesis

Study line: Information Systems Science

Page count: 73+0

Abstract: In software engineering, mutation testing is a method to assess and improve software test quality. It is more prevalent in object-oriented programming and has only seen little interest in functional programming.

In this thesis, leveraging mutation testing in functional programming is explored by constructing a practical mutation testing tool called *Mutix* for a functional programming language called Elixir. *Mutix* is published as an open source testing tool for any developer to use.

Keywords: Software Quality, Software Testing, Mutation Testing, Functional Programming

Suomenkielinen tiivistelmä: Mutaatiotestaus on ohjelmiston laadunhallintamenetelmä, jolla voidaan arvioida ja parantaa ohjelmistotestien laatua. Mutaatiotestaus on ollut suosituempaa olio-ohjelmoinnissa kuin funktionaalisessa ohjelmoinnissa, jossa sen kiinnostus on ollut vähäistä.

Tässä tutkielmassa luodaan konstruktio nimeltä *Mutix* funktionaaliselle ohjelmontikielelle nimeltä Elixir. *Mutix* julkaistaan avoimen lähdekoodin työkaluna, jota kuka tahansa ohjelmistokehittäjä voi käyttää ohjelmistotestiensä laadunhallintaa varten.

Avainsanat: Ohjelmiston Laatu, Ohjelmistotestaus, Mutaatiotestaus, Funktionaalinen Ohjelmointi

Glossary

AST	Abstract Syntax Tree.
Atom	Constant whose value is its name.
BEAM	Erlang Virtual Machine.
Developer	Software developer — a programmer who writes software.
ExUnit	Elixir’s built-in testing framework.
Local function	Function in the current context.
Macro	Code that writes code.
Metadata	Auxiliary data to describe or annotate other data.
Mix	Elixir’s built-in build tool.
Mix task	Defined action using behaviors defined by Mix.
Remote function	Function defined outside of the current context.
Runtime	Software system that executes code.
Run-time	Step at which code is executed.

List of Figures

Figure 1. Traditional mutation testing process	20
Figure 2. Stryker mutation score	24
Figure 3. PITest coverage report	24
Figure 4. Program lifecycle	34
Figure 5. AST visualization	36
Figure 6. AST visualization of a single node	36
Figure 7. Mutix’s AST transformer illustration	44
Figure 8. Test results from tests with mutations	48
Figure 9. Mutation report aggregated from test results	49
Figure 10. Mutation report & score.....	50
Figure 11. Feedback for surviving mutants	51
Figure 12. Performance improvement proposal.....	59

List of Tables

Table 1. Selected mutation libraries of each language ecosystem compared against a standard or unit testing library of the same language based on a GitHub (2023) search. Popularity is assessed in terms of GitHub stars given by developers. The <i>Popularity</i> column shows how many stars the mutation testing library has compared to the standard testing library of the same language.	29
---	----

Contents

1	INTRODUCTION	1
2	FAULTS IN SOFTWARE	3
	2.1 Static analysis	4
	2.2 Software testing	5
3	FOUNDATIONS OF MUTATION TESTING	6
	3.1 Mutation operators	7
	3.2 Competent programmer hypothesis	9
	3.3 Coupling effect	11
	3.4 Weak vs. strong mutants	12
	3.5 Equivalent and subsumed mutants	15
	3.6 Higher order mutants	17
4	MUTATION TESTING IN PRACTICE	19
	4.1 Mutation Score	22
	4.2 Problems with mutation testing	25
	4.3 Industrial adoption	28
5	FUNCTIONAL PROGRAMMING AND ELIXIR	31
	5.1 Elixir	32
	5.2 Metaprogramming	33
	5.3 AST and its manipulation	34
	5.4 Testing in Elixir	38
6	MUTIX	40
	6.1 Mutation operators	40
	6.2 Launching Mutix	41
	6.3 Injecting mutations	43
	6.4 Report	47
	6.5 Performance	52
	6.6 Caveats	54
7	DISCUSSION	56
	7.1 Limitations & future improvement	57
	BIBLIOGRAPHY	61

1 Introduction

As the world is digitalizing at a rapid pace the software engineering industry has enjoyed tremendous growth over the past decades. The code and version control hosting platform GitHub (2022) announced that there were over 3.5 billion new contributions on GitHub in 2022 alone, with over 227 million pull requests merged, adding new code to codebases.¹

The escalating rate at which new software is developed presents new challenges to maintaining proper quality control over the software. Over time, various techniques and methodologies have been developed to combat the issue, with software testing being the primary method. Ultimately, The IEE Computer Society that aims to document the industry standards came to designate testing a crucial part in the construction of software and the primary means to validate its correctness (Bourque, Fairley, and Society 2014, Chapter 3).

Even though it is well established in the industry that testing is a core component in software construction, surprisingly little attention has been paid to assessing the quality of tests. Most of the time in the software engineering industry companies rely heavily on static analysis as the primary guardian of quality, usually in the form of various human code reviews.

A testing technique developed in the 1970s called *mutation testing* aims to provide a solution to this dilemma. The idea of mutation testing is to introduce what are labeled mutants to a program, and find out how many of them the existing test suites catch and how many go undetected. Thanks to its role as a method for quality control for the existing software test suites mutation testing can be described as *tests for your tests*. This thesis explores the feasibility of mutation testing in a functional programming context where it has seen little interest so far.

1. Contributions mean commits, issues, pull requests, discussions, gists, pushes and code reviews.

More specifically, the research documented in this thesis aims to find out the answers to the research questions detailed below. In functional programming context:

- Can test quality be asserted by utilizing mutation testing?
- Can developer friendly feedback be extracted from a mutation testing tool?

In this thesis, the notion of *developer friendly feedback* is not defined strictly but is thought of as any feedback that is useful in helping the developer understand the context in which mutation testing finds problems in the software that is being tested.

To answer the research questions, this thesis initially focuses on the theoretical and technical aspects of mutation testing to lay a scientific foundation on the subject. Ultimately, As a solution, the creation of a new testing tool for the software's test suites is explored for a functional programming language called Elixir. The mutation testing tool will be released as an open source library, allowing any developer to utilize it to assess the quality of their existing software test suites.

The thesis is structured to first introduce the reader to software quality verification and the variety of testing techniques developed and how mutation testing has evolved alongside the more mainstream techniques employed by the industry. As the constructive solution to the research problem is being implemented in a functional programming language, there is subsequently a brief introduction to the paradigm as well as metaprogramming. Finally, an assessment is presented that discusses how well the constructed solution adheres to the original mutation testing concepts and also what new it brings to the table for traditional mutation testing.

2 Faults in software

Maintaining software quality has been a major industrial challenge in software engineering for decades. Technopedia (2020) defines programmer¹ as someone who writes computer software by providing specific instructions to a computer. As human centric work the quality of software is subject to the programmer's ability to make correct software. A data logging & analytics company called Coralogix (2015) studied developer productivity and found that a programmer creates an average of 70 bugs for every 1000 lines of code. Alargmingly, programmers were reported to spend the majority of their time debugging instead of developing new software.

Unexpected behavior and incorrect results in software engineering are called by varying terms in the industry. In this thesis the definitions of Williams (2010) that are based on the original "IEEE Standard Glossary of Software Engineering Terminology" (1990) are used:

- **Mistake** - human mistake that produces an incorrect result.
- **Fault** - an incorrect definition in the program.
- **Failure** - the system failing to meet one or more of its specified requirements.
- **Error** - the incorrect result of running the program as opposed to the specified or otherwise expected behavior.

In common terms, a human programmer makes a *mistake* which manifests as a *fault* in the program definition which usually is the program's source code. Faults are more commonly called *bugs* in software engineering. When executed, the fault becomes an *error* and ultimately a *failure* in the system can be observed.

Faults can be characterized as either syntactically or semantically. Offutt and Hayes (1996) describe common syntactic faults to be:

- Typos by programmer.
- Incorrect variable names.
- Incorrect implementation of the design specification.

1. *Programmer* and *developer* are used interchangeably in this thesis.

While syntax refers to the structure semantics in programming language theory means bringing meaning to syntactically valid programs (Floyd 1993). Offutt and Hayes (1996) characterize semantic faults as those that produce faulty outputs in some subset of inputs. The reason this distinction between syntactic and semantic faults is important is that it is used to approximate the size of the fault. That, in turn, is fundamental for example in projecting the effect various mutation operator categories have in mutation testing. Syntactic and semantic faults have some intersection, especially when a superficial syntactic change results in a large semantic fault. But as Offutt and Hayes (1996) note, most fault based testing techniques that mutation testing is a part of rely on syntactic fault injection and consider the faults generated syntactic by nature.

2.1 Static analysis

A static program analysis, often just called static analysis, describes methods for analyzing software programs without actually running them (Wichmann et al. April 1995). While a static analysis could mean a code review done by a human it typically means a technical, often automated way of conducting an analysis of the program based on its source code. Simplifying, static analysis can be considered the process of grammatic and spell checking for the program's source code (Williams 2010, Chapter *Static Analysis*).

Compilers and even runtime systems have various static analysis tooling built into them, asserting that certain kinds of errors do not exist in the source code before it is run. The most prominent static analyzers known to programmers in modern software development are type checkers in statically typed programming languages that validate that a type system is not violated. A type system is a set of logical rules that the programmer uses to prevent a class unwanted of program behaviors (Pierce 2002, Chapter 1.1). A type system is categorically able to prove the absence of certain behaviors. It is thus a way of eliminating a class of program faults without having to run the program.

Type systems thus provide assurances that certain types of errors cannot occur at run-time. For the programmer this means that such behaviors do not need to be tested at all by means of dynamic testing, eliminating potentially large amounts of test code that would otherwise

have to be written. For mutation testing this means if a mutation was to be injected that would violate the type system's rules, for example replacing a binary operator with a string concatenation operator the type system would already catch this while generating the mutated code if it violated the type system's constraints.

2.2 Software testing

In contrast with static analysis software testing is known as *dynamic analysis*. Dynamic analysis means evaluating the system or a part of it based on its behavior when executed (Williams 2010, Chapter *Static Analysis*). As the idea of software testing is running the program or its components to observe its behavior against expectations all software testing can be categorized as dynamic analysis. The key distinction is that while static analysis is done without executing the program or any part of it, running tests at run-time is crucial for performing a dynamic analysis.

Software testing can be categorized under two software testing umbrella categories - white-box and black-box testing. Black-box testing focuses on verifying the software's behaviour from the outside without analyzing or mutating its internal structures and logic. White-box testing, in contrast, targets the application's internal logic (Williams 2010, see sidebar in Chapter *Coverage Criteria*). Unit testing is a part of white-box testing and describes a level of testing where the functionality of software modules are verified in isolation from other elements (Bourque, Fairley, and Society 2014, Chapter 3.5.3). Mutation testing is most often built on existing unit test suites in order to verify the tests capture whenever the source code is infected with mutations.

3 Foundations of mutation testing

Mutation testing is a form of software testing that aims to assert the quality of an existing test suite that targets source code tests. In other words, it is a method to verify the quality of the existing test suites. As mutation testing targets the internal logic of the application rather than its functionality it falls into the white-box umbrella category of software testing (Williams 2010). Harman and Jia (September 2011, Chapter 1) categorize mutation testing more specifically as a fault-based testing technique. Sometimes mutation testing is further categorized as syntax-based testing due to the mutation operators applied being of syntactic nature (Ammann and Offutt 2017, Chapter 9.1.2). Mutation testing can be referred to as mutant testing or mutation analysis in the literature (Coles et al. July 2016).

As a concept, mutation testing originates from the 1970s when a student named Richard Lipton proposed the technique in a class term paper called *Fault Diagnosis of Computer Programs* (Offutt and Untch 2001). Demillo, Lipton, and Sayward (May 1978) further refined the concept later in the decade in their research which is now generally cited as the original reference for the foundations of mutation testing.

In their article, Demillo, Lipton, and Sayward (May 1978) argue that aside from rare occurrences such as errors originating from the operating system, errors in applications arise from:

- Missing control flows.
- Incorrect path selection in the control flows.
- Incorrect or missing actions.

These are all the result of a fault or multiple faults in the program's source code. Now, a test suite should normally catch all these faults before the program is deployed to production use. But there is inherently nothing that provides assurances of the test quality itself aside from simple coverage metrics which typically only detail the ratio of public interfaces of the source code tested by at least one unit test. Mutation testing was devised as a means to systematically and semi-automatically assert how well the existing test suite captures new faults introduced in the source code.

3.1 Mutation operators

Mutation testing tools create alternative versions of the program by injecting mutations into the original program. These infections are created by applying a set of predetermined rules which are called mutation operators¹ in mutation testing (Offutt and Untch 2001). Typically, these rules are replacements of operators with other syntactically valid operators, most often statically predefined as the industrial mutation testing tool documentation detail. However, despite the name mutation operators are not restricted to being replacements of operators specified by the programming language. They can range from replacing statements to erasing entire function bodies. As Ammann and Offutt (2017, Chapter 9.2.2) note, when designing mutation operators it is crucial to take the programming language context into account in order to successfully benefit from mutation testing.

In the scientific literature mutation operators are expressed in terms of what is referred to as *ground string* — any string that is grammatically valid for the program. In practice, ground string refers to the program that is to be infected and tested. Mutation operator is a rule that is applied to create syntactic variations that are likewise grammatically correct (Ammann and Offutt 2017, Chapter 9.1.2). The ultimate result is a mutant which is the output of applying a mutation operator once. Sometimes this process of injecting a mutant is called *mutagenesis* in the scientific literature (Petrovic and Ivankovic 2018).

Consider the following pseudocode example of calculating the arithmetic mean of two numbers given as arguments:

```
def average(a, b):  
    sum = a + b  
    return sum / 2
```

A typical mutation applied to this kind of algorithm would be converting the arithmetic `+` operator into its arithmetic opposite `-`. Now, imagine there was a test case asserting that:

1. Alternatively called *mutant operators*, *mutagens* or *mutation rules* (Wu et al. 1988).

```
test "average(a, b) returns the average of two numbers":  
  assert average(2, 6) == 4
```

After the mutation has been injected the statement `sum = a + b` would be transformed into a statement such that:

```
sum = a - b
```

This injected mutant would be a so-called first order mutant. Now when the same test case asserting the result of `average(2, 6)` is run the `average(2, 6)` function invocation would actually yield `-2`, making the assertion and subsequently the test case fail. In other words, this test would be able to capture the injected mutant. In the scientific literature this would be referred to as *killing* the mutant. Conversely, mutants that are not killed are often described as *surviving* mutants since they survive the test suite without being killed. In other words, a mutant survives when the test suite is run against a version of the application with the injected mutant with no test failing.

The scientific literature on mutation testing has taken the stance that since mutation operators have a syntactically light footprint they are referred to as syntactic mutations rather than semantic. This becomes quickly evident when browsing through the most cited research papers on mutation testing, most prominently the work of Harman and Jia (September 2011) as well as Offutt and Untch (2001). However, sometimes practical implementations go as far as removing the entire function body as the injected mutant operator (Niedermayr, Juergens, and Wagner 2016), essentially removing the entire semantics from the function. This goes to show that depending on the chosen mutation operators, infections can be introduced very flexibly to the program ranging from insignificant faults akin to human mistakes to semantically complex faults that alter the core logic of the program.

Industrially, a conservative set of mutation operators is typically used. At Google the following operators were applied in their experiment (Petrovic and Ivankovic 2018, Figure 3):

- Arithmetic operator replacement.
- Logical connector replacement.
- Relational operator replacement.

- Unary operator insertion.
- Statement block removal.

A look at the popular open source mutation testing frameworks available shows that most frameworks use the same operator categories as Google. That said, most frameworks extend the range of operators beyond what Google has used in their experiment, often with the aid of the features specific to the programming paradigm, language or platform:

- **Control flow statement** (Rafalko 2023, see *Mutators - Loop*; Hałas 2023, for example the *break continue replacement operator*; Schirp DSO LTD 2023, see `meta/if.rb`, `meta/break.rb` and `meta/case.rb` in the source code).
- **Object constructor call** (Coles 2023, see *Available mutators and groups*).
- **Error raising** (Rafalko 2023, see *Mutators - Exceptions*).
- **Regex** (Stryker Team 2023a, see *Supported mutators*; Schirp DSO LTD 2023, see `meta/regexp.rb` in the source code).
- **Constant replacement** (Coles 2023, see *Available mutators and groups*).

3.2 Competent programmer hypothesis

Fundamentally, mutation testing relies on two key assumptions: the *competent programmer hypothesis* and the *coupling effect*. Demillo, Lipton, and Sayward (May 1978) phrase the competent programmer hypothesis such that *programmers create programs that are close to being correct*. In that vein, it is thought that mutation operators should be simple and of syntactic nature as these are the kinds of mistakes a programmer would be likely to make. By the same token, reaching the correct program would require only minor changes, making it suitable grounds for mutation testing since it uses syntactic mutation operators whose semantic footprint is small. Deriving from the competent programmer hypothesis, mutation analysis by means of mutation testing should produce faults that are realistic. It is precisely this deduction that mutation induced faults should be realistic that is the subject of studies pertaining to the competent programmer hypothesis.

Since the entire premise of the competent programmer hypothesis is somewhat psychological instead of purely technical, most of the literature researching it is empirical by nature.

Findings by Andrews, Briand, and Labiche (2005) show that using common mutation operators to introduce first order mutations does produce faults similar to real ones, although they are harder to identify than faults added by editing the source code manually. Among the most prominent empirical research into fault realism, and by extension to the competent programmer hypothesis, is a frequently cited article by Just et al. (2014) that evaluated large open source codebases in terms of how mutant detection compares against real faults. The study was able to establish a substantial statistical correlation between the amount of detected mutants as opposed to the amount of detected real faults, independent of the code coverage for tests. Admittedly, the research made some questionable assumptions such as automatically quantifying an issue as a fault in the program if it appeared in a GitHub bug issue tracker. However, Kim, Kim, and In (August 2020) and Laurent, Gaffney, and Ventresque (April 2022) later came to the same conclusion in their practical experiments, verifying the statistical correlation of mutants with real faults in mutation testing.

However, there are some studies contradicting the findings that faults by mutation infections using syntactic operator mutations are correlated with real faults. Gopinath, Jensen, and Groce (2014) challenged the premise in their empirical research where by means of a regression analysis the authors benchmarked syntactic operator mutations in mutation testing across four different programming languages. This single study found that syntactic mutations are not in fact representative of realistic faults. Stein et al. (2021) arrive at what could be described as middle ground in their findings about the competent programmer hypothesis. The researchers utilized a novel AST introspection approach to identify bugs statically in the source code rather than by simply running test suites and comparing their results. While the study in principle confirms the validity of the assumption, they find that the lack of appropriate operators is most often responsible for the failure to generate realistic faults. Ultimately, this inconclusiveness in the scientific community leads to question if there is enough scientific evidence to conclusively confirm the correlation of syntactic mutations with real-world faults. This in turn casts some doubt on using the competent programmer hypothesis as one of the core premises in mutation testing when it comes to designing mutation operators.

3.3 Coupling effect

In their original article Demillo, Lipton, and Sayward (May 1978) had the thought that a test that is able to uncover a simple error is also capable of uncovering a more complex, related error. This was labeled *the coupling effect*, denoting simply that complex errors are implicitly always coupled to simple errors.

Although the notions of simple and complex fault are not always so evident in software engineering it is thought that subtle faults are generally less explicit and subsequently harder to discover and pinpoint than simple faults like what first order mutation infections typically produce (Morell 1988). Mutation testing leverages the coupling effect by defeating most of the complex faults with test suites that are structured to kill simple mutants.

The coupling effect hypothesis first gained empirical support decades ago when Offutt found that a test suite that detects all simple faults will also be able to detect a large portion of the complex faults in the same program (Offutt 1989, January 1992). Offutt defines simple faults as single changes to the source statement while complex faults require more than a single change. Sometimes this simple versus complex fault distinction is defined in terms of order of mutations where a first order mutation is an injection of a simple fault and a second or higher order mutation is an injection of a complex fault or a set of faults (Offutt 1989).

While Offutt (January 1992, page 6) notes that the coupling effect is by nature probabilistic rather than absolute, the hypothesis has gained some theoretical support by means of a mathematical analysis by How Tai Wah (2001a). However, in contrast with the earlier empirical findings the analysis found that while the coupling effect occurs, it does so infrequently. How Tai Wah pursued additional research that validates that tests that can kill first order mutants are able to kill most of the higher order mutants (How Tai Wah 2001b, 2003). In essence, the coupling effect is established both theoretically and empirically and it seems to be less disputed in the scientific community than the other key assumption in mutation testing — the competent programmer hypothesis.

3.4 Weak vs. strong mutants

For a test suite to successfully catch and kill mutants in mutation testing, certain premises need to be fulfilled. Offutt and Untch (2001) touch on these requirements to make mutation testing feasible, defining three conditions ²:

- **Reachability condition** — the test must reach the mutated statement.
- **Necessity condition** — the mutated statement must be executed by the test and lead to an error in the program's state.
- **Sufficiency condition** — the incorrect state must propagate in the program to result in an incorrect output.

Deriving from the definitions of *fault* and *failure* from chapter 2, the conditions presented above are generalized by Ammann and Offutt (2017, see Chapter 2.7 about the origins) into a *RIPR* model that stands for *Reachability, Infection, Propagation* and *Revealability*. The *RIPR* model is defined as a universal model for software testing, not just for mutation testing. According to the model, the test must first reach the faulty location (*Reachability*). As the faulty location gets executed it must result in a failure — an incorrect state for the program (*Infection*). The infection must then propagate in the program to result in an incorrect final state (*Propagation*). Ultimately, the test must be able to discover the propagated incorrect final state and output of the program (*Revealability*).

Defining mutation testing in terms of the *RIPR* model is straightforward. A mutant would cause an infection that would need to be reached by the test. The execution of the mutant (the mutated code) would then cause an incorrect state inside the program, resulting in an eventual incorrect output or final state. If the test discovers the mutant, in other words fails one or more assertions, it gets *killed* in mutation testing terminology. This is how the standard mutation testing technique, which is sometimes called *strong* mutation testing to make a distinction between *weak* mutation testing, applies the *RIPR* model.

Building on his work on algebraic testing, Howden had proposed the concept of *weak* mu-

2. As mutation testing was originally implemented in imperative languages the above definitions are defined in terms of imperative statements. With functional or declarative languages the necessity condition would imply that a mutated expression would have to be evaluated for the condition to be fulfilled.

tation testing in an effort to reduce the cost of mutation testing (Howden March 1978). In weak mutation testing, in order to kill a mutant only the reachability and infection conditions need to be met. As opposed to strong mutation, testing with this relaxed definition of killing a mutant the propagated incorrect output does not need to be detected by a test for the mutant to be killed.

As Offutt and Lee (1991) remark, mutation testing is primarily implemented as a unit testing technique the practical difference between weak and strong mutation testing is disputable. Unit tests, by nature, assert individual functions or smaller parts of the program rather than the program's holistic behavior. Since unit tests are run without the full context of the program to keep their scope minimal there is limited space for an incorrect state caused by an infection to propagate. Moreover, unit tests generally do not even attempt to assert faulty state propagation within the larger program context. That said, unit tests often do assert the output of their test target like a function invocation which could be argued fulfills the propagation and revealability aspects of the RIPR model. As a matter of fact, Howden (1982) himself fails to offer a precise definition for a *component* in his paper that the definition of weak mutation uses as a subset of a program that is tested. This makes it hard to reason about the precise differences of weak and strong mutation testing in practice.

The definition for weakly killing mutants used by Williams (2010, Chapter 9.2.2) delineates that the state of the execution of the program needs to be different immediately after the execution of the mutant as opposed to the original program. In contrast with strongly killing a mutant, in weak mutation testing the infected program output does not necessarily need to be different, or at the very least it does not need to be validated by a test. As a more precise definition this gives room to explore a practical example for weak mutation testing. Consider the following Javascript function that doubles the input number while keeping track of the doubled integer count:

```

var counter = 0

function doubleWithCounter(x) {
  if (Number.isInteger(x)) {
    // mutate below to: counter = counter - 2
    counter = counter + 1
    return x * 2
  } else {
    return x
  }
}

```

As can be seen from the commented line the statement incrementing the counter will be mutated in a mutation testing suite. Now, the reachability condition for the mutant is $x \in \mathbb{Z}$, that is to say the execution will reach the mutant for all integers passed in as an argument. A test case asserting the counter incrementing would now confirm the reachability condition is satisfied and the statement executed, validating the infection condition from the RIPR model as well:

```

test('increments counter with integer arguments', () => {
  expect(counter).toEqual(0)
  doubleWithCounter(2)
  expect(counter).toEqual(1)
})

```

In a mutation test the above test case would fail the assertion since `counter` would actually be `-2` at the second assertion, thus successfully killing the mutant. However, as the output of the function was neither asserted nor actually altered by the mutation it leaves the propagation and ultimately its revealability conditions unsatisfied. Therefore, the mutation test performed on this source code with any integer would only result in weak mutant killing, hence weak mutation testing. To transform the test into a strong mutation test it would have to satisfy both of the propagation and revealability conditions on top of the reachability and

infection conditions that weak mutation testing is also required to satisfy. To do that in this example, the mutation infection should alter the returning statement, for example, and the test case should assert the return value rather than the mutated counter value.

Since the weak mutation technique was originally conceived in the hopes of lowering the computational cost of mutation testing, there have been multiple empirical studies over the years, benchmarking weak mutation testing against standard mutation testing. For example, Offutt and Lee (1991) found that weak mutation testing is best applicable to small components with simple mutations Papadakis, Malevris, and Kintis (January 2010) tracked a weak mutation test suite to catch a 97% of the strong mutants while reducing the need for mutant generation by 73% in certain circumstances. Ultimately, as evidenced by the studies mentioned, weak mutation testing relies too tightly on programming language and runtime implementation details to claim as a universally effective method in combating high computational costs related to mutation testing.

3.5 Equivalent and subsumed mutants

Equivalent mutants have been a persistent problem in mutation testing for decades. Frankl, Weiss, and Hu (1997) define equivalent mutants as mutants who *compute precisely the same function as the program*. In other words, equivalent mutants are mutants that while inject a mutation to the original program still result in a semantically equal program. As equivalent mutants change no program logic, only syntax, they are impossible to detect using first order mutation testing.

The reason equivalent mutants are problematic is because a mutation testing tool injects a mutant into the program's source and then expects the tests to kill the mutant. But since the program, despite the mutant infection, behaves precisely as it should under normal circumstances the existing tests naturally pass without detecting the mutant. In this way it creates false positives in the mutation score that is detailed in section 4.1.

For example, consider injecting an equality operator mutation to the following algorithm:

```
def max(a, b, c):  
    if (a > b && a > c) return a  
    elif (b > c) return b  
    else return c
```

For equality operators, the commercial Stryker mutation testing tool would create a mutant of the `>` operator, mutating it into `>=` (Stryker Team 2023a, see *Supported mutators*). With a single order mutation infecting a single statement mutating the `elif`-statement we would get the infected conditional statement:

```
elif (b >= c) return b
```

As this infected code would behave exactly like the original code does for all possible inputs no test could detect the infection, thus resulting this infection to be an equivalent mutant. Some industrial tooling have taken the stance to explicitly educate the programmer about the possibility of getting false positives that are in fact equivalent mutants (Stryker Team 2023a, see *Equivalent mutants*).

It should be noted that it is generally very difficult to assess if a mutant survives because the infected code is semantically equal to the original code or because there is a test case missing that could detect the mutant. A test suite would virtually never test every function and all logic in the program with all possible inputs that would traverse all possible logical code paths at run-time. As Parsai and Demeyer (2018, see *Introduction*) note, mutant subsumption is thus often used as a compromise, more specifically dynamic mutant subsumption.

Subsumed mutants are mutants which are subsumed by other mutants. Parsai and Demeyer (2018) define mutant subsumption using the following example: *mutant A truly subsumes mutant B if and only if all inputs that kill A also kill B*. To add, a *dynamic* mutant subsumption happens when *mutant A subsumes mutant B with regards to test set T if and only if there exists at least one test that kills A, and every test that kills A also kills B*. A dynamic mutant subsumption is therefore a compromise, basically declaring a mutant subsuming another mutant as long as this holds true in the context of the test suite. Otherwise to establish that

a mutant subsumes another mutant all possible inputs would have to be tested via exhaustive mutation testing which in most situations is simply not practical. From a testing standpoint subsumed mutants are completely redundant because killing the host mutant will always likewise kill the subsumed mutant. Subsumed mutants therefore unnecessarily inflate the mutation score introduced in section 4.1.

There has been both scientific and industrial work to combat the issue of subsumed mutants or rather utilize the phenomenon to reduce the mutant count. Ammann, Delamaro, and Offutt (2014) proposed a theoretical way to identify the minimum number of mutants needed in a given test suite that allows retaining confidence in the given test suite. Building on their research, Papadakis et al. (2016) present evidence in their research that subsumed mutants not only are redundant in terms of testing but can in fact pose threats to test validity, highlighting the need to take subsumed mutants into account while doing mutation testing. Finally, to put the prior research into practice, Parsai and Demeyer (2018) introduced *LittleDarwin* — a Java testing tool that drastically reduces the number of mutants created by means of dynamic mutant subsumption.

3.6 Higher order mutants

As mutation testing has seen various implementations in the industry over the years the science has also evolved alongside it. Purushothaman and Perry (July 2005) found that there is only a mere 4% chance that changing a single line in the source code will introduce a fault in the code. With the aim of reducing the amount of redundant mutants created Offutt (January 1992) first brought up the idea of applying mutation operators twice, calling it *second order mutation testing*. Second order mutation testing was further studied by Polo, Piattini, and García-Rodríguez (June 2009), Kintis, Papadakis, and Malevris (April 2014) and Madeyski et al. (2014). The studies found that by using specific strategies it is possible to sometimes cut the number of tests that need to be run by as much as half by combining two mutants into one mutation.

In 2010 Harman, Jia, and Langdon (September 2010) published a paper founded on the premise of second order mutant testing, proposing a new paradigm called *higher order mu-*

tation testing. Basically, in higher order mutation testing, mutants are composed from the combination of two or more mutants at the same time. According to the proposal, to locate the higher order mutants while filtering out irrelevant or unrealistic mutants or mutants that are technically impossible to kill, a search algorithm based technique is used. The research was done to establish a scientific foundation for a C language higher order mutation testing tool called *MILU* the authors had constructed two years prior (Jia and Harman 2008).

What is also novel for mutation testing in the research by Harman, Jia, and Langdon (September 2010) is how a new fitness (fitness for purpose) function is introduced whose purpose is to evaluate the quality of each mutant. The quality is determined by the syntactic and semantic footprint of the mutant as well as how hard it is to kill. In practice, this means that a mutant is rated high quality if it is either:

- **Single-objective** - harder to kill than a corresponding first order mutant.
- **Multi-objective** - syntactically and semantically less significant than other mutants.

Single-objective mutants are basically all subsuming mutants. While syntactic and semantic insignificance as a criteria for a higher quality mutant in the multi-objective approach might sound counter-intuitive, it merely means that the mutant is hard to detect and is thus a more valuable target to find for automated software test tooling.

Generally, studies explore various combinatorial strategies whose aim is to identify the most effective methods in suppressing the redundant mutant count while still maintaining a high confidence in the mutation test coverage (Papadakis and Malevris April 2010; Papadakis, Malevris, and Kintis January 2010). In essence, the studies attempt to find the sweet spot between cost and effectiveness to make second or higher order mutation testing industrially viable. Concluding based on multiple studies, Van Do, Thi, and Nguyen (2014) state that higher order mutation testing is still the most promising approach to reduce the amount of mutants created. However, it has not enjoyed the same popularity in the industry thus far. There are very few prominent tooling available that leverages higher order mutation testing. It remains to be seen if the industry will gain commercial interest in developing testing tooling for higher order mutation testing or not.

4 Mutation testing in practice

A testing process is a distinct process from mutation testing. A testing process is a sequence of steps to take to produce actual test cases (Ammann and Offutt 2017, Chapter 9.2.2; Offutt and Untch 2001). On a high level Williams (2010, Chapter *White-box Testing - Deriving Test Cases*) calls this process a *test design process* which he rates as vital as executing the test cases themselves. While software testing processes are well established by literature the actual process for mutation testing differs substantially from the conventional processes. This is because as Offutt and Untch (2001) describe, mutation testing is a method to assess the quality of the existing test suite. The actual testing of the software is therefore just a side effect of the mutation testing process.

Mutation testing tools start by ingesting the program to be tested as an artifact which is virtually always its source code. The tool then creates mutants by applying mutation operators on the source code. Sometimes at this point there is an additional stage of mutant reduction that aims to reduce the resource consumption at run-time in mutation testing by eliminating a portion of the mutants to be created with some heuristic (Ammann and Offutt 2017). Next, the tool receives the existing test suite and typically executes that against the original program to validate the premise ready for mutation testing. In other words, the existing test suite must pass without errors to validate it is correct.

When the test setup is confirmed valid for mutation testing the testing tool executes the test suite against each mutant, in other words a mutated version of the application. As Offutt and Untch (2001) note in their description of the mutation analysis process the key goal of running the test suite is to distinguish the mutated program from the original. Whenever running a test case on the mutated code fails an assertion or raises an error at run-time that mutant gets marked as killed and excluded from subsequent test cases (Ammann and Offutt 2017, Chapter 9.2.2, see *Testing Programs with Mutation*). In other words, for a mutant to be killed by the mutation test, the test case has to fail where it normally would not.

Once the mutation testing tool has run the full mutation test suite the tool aggregates a coverage report called *mutation score* which is properly introduced in section 4.1. Ideally, the

mutation score should be 100%, signaling that all mutants have been successfully killed by the test suite although this is rarely the case. Therefore, as Ammann and Offutt (2017, Chapter 9.2.2) explain the programmer or tester typically compares the resulting mutation score against a threshold value which is contextually the minimum accepted mutation score.

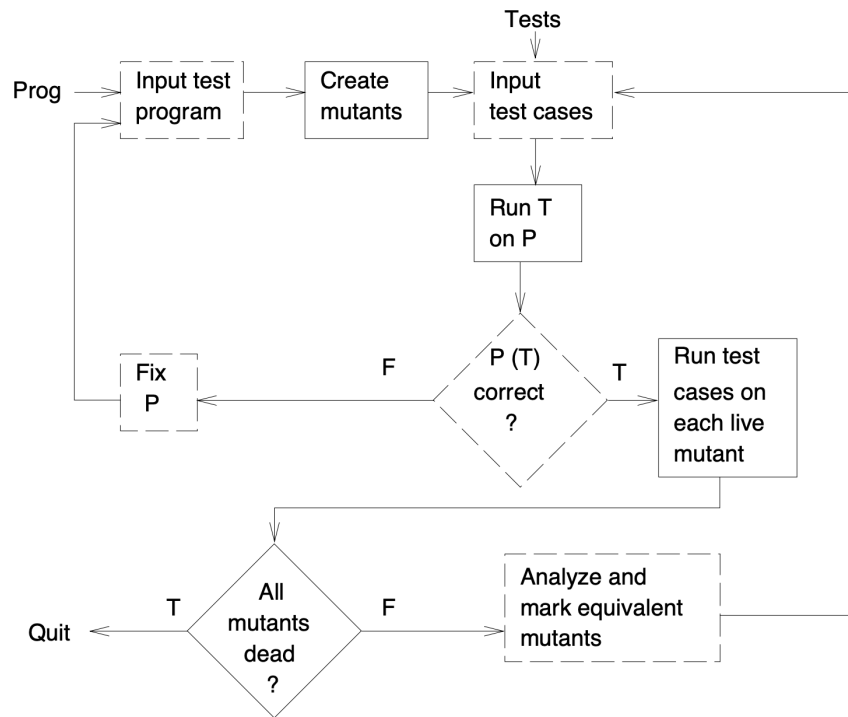


Figure 1. The traditional mutation testing process as presented by Offutt and Untch (2001). Solid boxes denote automated steps while dashed ones highlight steps that require manual programmer input.

As can be seen from the figure 1, in traditional mutation testing the existing software test suite is run on the program first to assert there are no failures in the test suite. If any failure arises, it needs to be fixed so that all tests run successfully. After this, the test suite is run against mutated versions of the program. As an iterative process, for as long as there are mutants that survive, the programmer will need to keep adding tests or refactoring the existing suite, verifying each time that they run successfully against the unaltered source program. As most mutation testing tools are unable to detect equivalent mutants, they are required to be marked as ignored in the testing process so that they are no longer considered surviving mutants.

Once all mutants have been found, or optionally when a predetermined, accepted threshold for mutation score is achieved the mutation testing process concludes. With this process, the programmer is able to assert and improve their test suite effectively, with manual inputs in the iterative loop being:

- Fixing the original tests or the source program to get all tests to run successfully.
- Marking equivalent mutants.
- Adding code coverage with new tests or by improving existing tests to capture all non-equivalent mutants.

4.1 Mutation Score

The main purpose of a software test suite is to detect and provide feedback of what faults were found and where in the source code. Testing frameworks usually aggregate simple coverage reports of test suites as feedback, most commonly reporting *code coverage* for the tests. In mutation testing there is a much more specific report produced from running the mutation test suite. This is called *mutation score* and its main idea is to detail the amount of mutants killed from the total amount of mutants discovered (Ammann and Offutt 2017, Chapter 9.1.2). Formally, mutation score is defined by Offutt (January 1992) as the following:

$$\text{mutation score} = \frac{\text{number of killed mutants}}{\text{number of all mutants} - \text{found equivalent mutants}}$$

In practical terms, this mutation score formula calculates the percentage of mutants killed by the test case or suite that are not equivalent. Equivalent mutants, which are mutants that are semantically equivalent with the original program, cannot by nature be killed as they behave exactly as the program correctly should. With a 100% mutation score all non-equivalent mutants would've been killed by the mutation test suite. Such a test suite that kills all non-equivalent mutants is described *adequate* in literature (Offutt and Untch 2001; Parsai and Demeyer 2018; Ammann, Delamaro, and Offutt 2014).

Instead of just reporting the plain mutation score itself, various mutation testing tools used in the industry aggregate and display various other relevant information in their mutation score reports. Akin to typical code coverage reporting tools, PHP's `infection` testing tool by Rafalko (2023) reports code coverage but for for mutation tests:

$$\begin{aligned} \text{TotalCoveredByTestsMutants} &= \\ &\text{TotalMutantsCount} - \text{NotCoveredByTestsCount} \\ \text{TotalDefeatedMutants} &= \\ &\text{KilledCount} + \text{TimedOutCount} + \text{ErrorCount} \end{aligned}$$

$$\begin{aligned} \text{CoveredCodeMSI} &= 100 * \\ &(\text{TotalDefeatedMutants} / \text{TotalCoveredByTestsMutants}) \end{aligned}$$

Instead of simply computing the mutation score, this formula combines it with code coverage to produce something useful — the mutation score for source code that is actually covered by tests. This gives a more realistic approximation of how effective the existing tests really are. Eventually, the tool aggregates the following kind of report from running the mutation test suite:

```
2 mutations were generated :
  2 mutants were killed
  0 mutants were configured to be ignored
  0 mutants were not covered by tests
  0 covered mutants were not detected
  0 errors were encountered
  0 syntax errors were encountered
  0 time outs were encountered
  0 mutants required more time than configured
```

Metrics :

```
Mutation Score Indicator (MSI): 100%
Mutation Code Coverage: 100%
Covered Code MSI: 100%
```

Often mutation testing tools further break the report down by test suite or type while packing in various other details such as error types while executing the suite. The commercial `stryker-js` by Stryker Team (2023b) looks like the following:

File / Directory	Mutation score	Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
All files	57.72	86	63	0	0	0	10	0	86	63	159
components	60.71	68	44	0	0	0	10	0	68	44	122
pipes	61.54	8	5	0	0	0	0	0	8	5	13
services	57.14	8	6	0	0	0	0	0	8	6	14
router.js	20.00	2	8	0	0	0	0	0	2	8	10

Figure 2. Mutation Score as generated by the `stryker-js` mutation testing framework (Stryker Team 2023b).

Beyond just statistical coverage reports some mutation testing tools offer more detailed reports of the mutation coverage in the source code. For example, Java's `PITest` provides a graphical report combining line and mutant coverage:

```

122 // Verify for a "." component at next iter
123 if ((newcomponents.get(i)).length() > 0)
124 {
125     newcomponents.remove(i);
126     newcomponents.remove(i);
127     i = i - 2;
128     if (i < -1)
129     {
130         i = -1;
131     }
132 }
133

```

Figure 3. Java's `PITest` coverage report (Coles 2023). Light green and light pink indicate line coverage and the lack of it while dark green and dark pink denote the mutation coverage and the lack of it.

While science explores and formalizes new methods, paradigms and patterns it is typically left to the industry to solve the practical issues that come with them. These open source tools highlight how each tool has found their own way of providing useful feedback to the developer beyond what is defined in the scientific literature for mutation testing.

4.2 Problems with mutation testing

While mutation testing as a technique is practically half a century old concept it has seen surprisingly few practical applications industry-wide. In their article in the early 2000s Offutt and Untch (2001) argued that that this is due to three primary reasons:

- Exhaustive testing is often not economically sensible to implement.
- The industry failing to successfully integrate unit testing into the software engineering process.
- Lack of sufficiently automated tooling.

Given that in terms of processing power computers decades ago were a far cry from what they are today it is understandable that a mutation test suite generating a large amount of additional tests to be executed presented a major bottleneck hindering the adoption of mutation testing.

There has been some research on reducing the run-time costs of mutation testing, such as investigating means to reduce run-time costs on Java by Falah, Salwa, and Achahbar (September 2013). Typically, the studies investigate reducing the number of mutants injected so that there are simply less tests to execute at run-time, taking less computational resources. As one injected mutant presents one syntactic fault, with a small semantic footprint in the program a vast number of mutants needs to be created to achieve sufficient coverage of code paths that could fail. For example, to achieve full coverage on a test suite of 100 test cases in a program that has 150 mutants injected to it would require a total of $100 * 150 = 15,000$ test executions. To make matters worse, some empirical findings indicate that only some 5% of the generated mutants are useful (Papadakis et al. 2016).

The scientific community has attempted to formalize methods to combat the high computational cost of mutation testing. As the discovery of realistic and thus relevant mutants is of probabilistic nature Sahinoglu and Spafford (July 1999) attempted to model a sampling solution using a Bayes sequential procedure — a probability ratio test. Using this approach, the study claims it is possible to drop the amount of test cases an order of magnitude lower, effectively to 1-10% range of the otherwise deterministically executable amount of tests required for traditional mutation testing without sacrificing confidence in the test suite.

Now with the emergence of open source machine learning tooling available, in the past decade the scientific community has even made attempts to work their way around the high computational costs by adopting predictive machine learning models. Since the discovery of mutants can be classified binary - the mutant is either killed or survives - Zhang et al. (2016) proposed training a machine learning model that attempts to discover mutants without actually executing any mutants. This was implemented using a *random forest* decision tree algorithm as proposed by Breiman (October 2001) and found to be an effective solution detecting mutants without exhausting too much computational capacity. Ultimately, despite the recent advances in reducing the mutation count Papadakis et al. (2019) conclude in their comprehensive analysis of the state of mutation testing in 2019 that the issue of generating too many mutants still remains a crucial and a largely unsolved issue.

As for the lack of unit testing practices in the industry, this is probably outdated information as unit testing is a common practice today. At Google, for instance, most tests are written as unit tests (Winters, Manshreck, and Wright 2020, Chapter 12). Google guides its engineers to craft mixtures of 80% unit tests and 20% broader scoped tests. Moreover, some like Zilberfeld (March 2014) argue that unit testing is essential to the *agile software development* process, directly contradicting the Offutt's and Untch's claim that unit testing is not well integrated into the software development process. With the emergence of *Continuous Integration & Continuous Delivery* methodology along with its numerous practical continuous integration implementations it could be argued that tooling automation is hardly an issue today anymore either (Synopsys 2023).

Despite the recent advancements in hardware and the substantial progress made researching mutation testing the technique still suffers from the equivalent mutant problem (Papadakis et al. 2019, Chapter 5.3.1). As explained in section 2.2 software testing is a form of dynamic analysis that evaluates the result of running the program or parts of it. This means that in order to verify that an infected program is semantically equal to the original program to prove the mutant's equivalence, tests are required to cover every code path with a vast range of accepted inputs.

Various heuristics have been developed over the years which typically are optimization techniques. A systematic literature review concluded that higher order mutation testing, as ex-

explored in chapter section 3.6, is still the most promising method for solving the problem of equivalent mutants. The study included an empirical experiment that produced modestly positive results in reducing equivalent mutant generation, although at the cost of slightly decreasing overall mutation test quality (Madeyski et al. 2014).

Another approach to alleviate the equivalent mutant problem could be a form of static analysis that compares the original program against the mutated variety. Such a solution would require a thorough understanding of the programming language in question in order to reason about its syntax and semantics. Papadakis et al. (2019, Chapter 7.1) discuss the idea of using behavioral models from the likes of automata theory to formalize the issue as a language equivalence problem. Other contemplated solutions employing static analysis are static symbolic execution of mutants as well as static data-flow analysis which have seen some preliminary research about them (Papadakis et al. 2019, Chapter 5.3.1).

Although to solve a different problem in mutation testing, Stein et al. (2021) experimented with a path finding approach in mutation testing using abstract syntax trees (ASTs¹). Such an approach could theoretically be used to search for mutations that are semantically equivalent in the context of the program. In fact, sometimes generic compiler optimization techniques are leveraged when investigating the AST semantic difference. Compiler optimizations have been a subject of intensive research and as a result, modern compilers are optimized to compile the source code into the most efficient form of run-time code. Papadakis et al. (2015) found that due to the compiler optimizations sometimes the compiled run-time code of the original program is exactly the same as it is for the infected program. This means that the compiler itself can be used to weed out the equivalent mutants whenever they arise. The same empirical study found that a specific compiler optimization technique that was deployed was able to detect up to 30% of the existing equivalent mutants.

Contrary to the studies regarding the competent programmer hypothesis Van Do, Thi, and Nguyen (2014) argue that mutation testing also suffers from lack of realism, claiming that single application of syntactic mutation operators do not produce realistic faults. Their proposal was based on the suggestion of Langdon, Harman, and Jia (2010) that 90% of the real faults occurring are of complex nature, meaning the source code would need to be changed

1. Skip to section 5.3 for an introduction to *Abstract Syntax Trees*.

in multiple places to beat the fault. Higher order mutation testing was seen as the primary solution in fighting the lack of realism because the idea of higher order mutation testing is to produce complex faults which are considered more realistic by the authors.

Aside from problems more specific to mutation testing that are hindering its adoption in the industry mutation testing still suffers from the same issues as traditional unit testing does, like non-deterministic test execution results that in mutation testing propagates to non-deterministic mutant discovery (Harman, Jia, and Langdon September 2010, Table 1).

4.3 Industrial adoption

Some large corporations have experimented with mutant testing in industrial use at scale. Citing high computational costs with traditional mutation testing, engineers at Google experimented with a novel probabilistic approach to drastically limit the number of mutants to be injected (Petrovic and Ivankovic 2018). Google’s engineers managed to reduce the amount of mutants generated by traversing the AST and skipping *mutagenesis* for uninteresting nodes like logging statements. The solution was deployed at scale, being utilized by 6000 engineers affecting a total of 13,000 developers in the review process. Ultimately, the experiment reported discovering surviving mutants ranging from 1% to 13.2% depending on the programming language. With 75% of the developers rating the experiment useful mutation the authors pledged to continue working towards better mutation testing solutions for the industry.

As GitHub (2022) reported over 90% of the companies are currently using open source and 30% of the Fortune 100 companies are maintaining their own open source projects. The ecosystem of open source libraries and tooling evidently plays a major part in industrial adoption of scientific solutions. To get a gauge of the popularity of mutation testing among the open source libraries, the most popular testing solution for each programming language is compared against a selected standard or unit testing counterpart in table 1.

Language	Mutation library	Testing library	Popularity
OOP / Multi-paradigm			
C / C++	700 mull	17k catch	4%
Python	308 MutPy	17k pytest	3%
Java	1.5k PITtest	5k JUnit 4	18%
JavaScript	2.4k stryker	42k jest	6%
PHP	1.9k Infection	19k PHPUnit	10%
Ruby	1.9k mutant	3.1k minitest	60%
Rust	595 mutagen	<i>built-in</i>	
Swift	438 muter	9.7k Quick	
Functional / LISP			
Haskell	3 mucheck	716 hspec	<1%
Clojure	100 Mutant	<i>built-in</i>	
Ocaml	38 mutant	<i>built-in</i>	
Racket	6 mutate	17 rackunit	35%

Table 1. Selected mutation libraries of each language ecosystem compared against a standard or unit testing library of the same language based on a GitHub (2023) search. Popularity is assessed in terms of GitHub stars given by developers. The *Popularity* column shows how many stars the mutation testing library has compared to the standard testing library of the same language.

While a simple GitHub search is by no means a scientifically sound assessment of the popularity of various open source testing tools it does give a rough idea of how mutation tools have made their way into the open source industry. For functional or LISP derivative programming languages, there appears to be very little practical interest for mutation testing in the open source community. For the popular mainstream object-oriented languages, however, mutation testing appears a fairly well known testing technique. Some of the mutation testing tooling is even introduced in scientific publications like Java’s PIT (Coles et al. July 2016).

Although nothing is inherently preventing implementing a mutation testing library or frame-

work in a functional programming language as can be seen from the table above there appears to be very few prominent, open source libraries available for mainstream functional programming languages.

5 Functional programming and Elixir

Mutation testing, while mostly studied in the context of object-oriented programming languages, places no restrictions on the programming language paradigms or their features. Programming has evolved to employ different programming paradigms and functional programming languages implement the functional programming paradigm (Hudak September 1989). As Hudak (September 1989, p. 363) describes, while functional programming languages were originally strongly motivated by the introduction of the lambda calculus by Alonzo Church in the 1930s, most functional programming languages do not implement it strictly.

Although functional programming languages often have constraints unlike those typically found in imperative languages, such as immutable data structures, mutation testing can be technically implemented in many different ways. Exploring the codebases of the open source mutation testing tools detailed in table 1, it can be quickly seen that there is no consistent technical pattern employed by the tools — sometimes they choose to use common object-oriented abstractions like dynamic dispatch while others decide to use macros and metaprogramming¹ to implement mutation testing, much like the construct in this thesis.

That said, the prevalence of mutation testing in functional programming in scientific publications is scarce. There are only fairly random references or benchmarks done for mutation testing with functional languages, such as Gopinath, Jensen, and Groce (2014) using Haskell as one of the benchmarking languages for empirically validating the *competent programmer hypothesis*. Since mutation testing seems largely under the radar for functional programming language communities, as evidenced by table 1 discussing the industrial adoption of mutation testing, this thesis introduces a new tool for mutation testing for a functional programming language called Elixir.

1. For example, see: <https://github.com/llogiq/mutagen#how-mutagen-works>

5.1 Elixir

Elixir is a functional, dynamically typed programming language originally developed by José Valim but now maintained by many other people as well (The Elixir Team 2023). It compiles code to run on the Erlang Virtual Machine, a runtime² often abbreviated as BEAM, which is famous for implementing the actor concurrency model. Erlang was first introduced in 1986 but is still actively maintained and developed (Armstrong 2007).

As a programming language, Elixir language has the following notable features (The Elixir Team 2023):

- **Immutability** — all data structures are immutable.
- **Expressions** — everything is an expression.
- **Pattern matching** — for asserting if specific values present in an expression.
- **Metaprogramming** — support for compile-time programming.
- **Polymorphism** — by using an abstraction called protocols.

Since Elixir and Erlang share the same runtime, Elixir was engineered to have good native support for the Erlang language as a key benefit of using Elixir (Elixir School 2023, see Erlang Interoperability). With the built-in interoperability, Erlang standard library as well as 3rd party libraries can be used directly in Elixir code. The construct in this thesis, however, has no dependencies to any 3rd party library or codebase so this interoperability is not used.

Notion commonly used in this thesis to denote functions includes the function name and *arity*, for instance: `sum/1`. Elixir’s source code’s documentation describes *arity* as *the number of arguments the function is to be called with* (The Elixir Team 2023, see `Functions` module and the `info/1` documentation there). Sometimes the source module in which the function is called from is added to the notion, for example: `Math.sum/1`. At times, function calls from other module’s functions than in the context are called remote calls and they’re referred to as such in this thesis.

The next sections focus on introducing the compilation process in Elixir as well as metaprogramming since both the compiler as well as metaprogramming are leveraged in the mutation

2. Not to be confused with the step at which code is executed — run-time.

testing tool created in this thesis. Other notable features, like pattern matching, are only used as an abstraction in the construct’s source code to define expressions in an idiomatic way typical to Elixir codebases but not as a pivotal pattern without which the construct could not be built.

5.2 Metaprogramming

Unlike many other programming languages, Elixir exposes an internal representation of code for programmers to programmatically inspect and change the semantics in the compilation artifact by means of metaprogramming. Metaprogramming is an ability with which computer programs are able to represent programs as data (Domkin 2021). Conceptually, metaprogramming was first made popular by LISP already in the 1970s but has since found its way to some other languages, with Elixir being one of them. Metaprogramming is used as a key method to implement mutation testing in the construct in this thesis. Specifically, it is used as the method to inject mutations into the source code using a feature in metaprogramming in Elixir that Iyengar (2023, p. 155) calls *dynamic code injection*. This process is described in detail in section 6.3.

The fundamental definitions of metaprogramming in the literature, particularly concerning Elixir, vary from abstract notions to practical descriptions of how code gets written. Iyengar (2023, p. 151) makes the following distinction: *writing a simple piece of code is referred to as programming, and writing Elixir code that programmatically injects behavior into other Elixir code is referred to as metaprogramming*. Elixir School (2023), on the other hand, describes metaprogramming simply as *the process of using code to write code*. Regardless of the somewhat different descriptions for metaprogramming, both the Elixir School and Iyengar describe metaprogramming as a feature that allows extending Elixir’s own features and behaviors and improve developer productivity. Therefore, it is often used to reduce repetitive boilerplate code in Elixir codebases. As a matter of fact, metaprogramming is so widely used in Elixir’s own source code that Iyengar (2023, p. 156) goes as far as stating that *most of Elixir is written in Elixir itself*. This thought is shared by McCord (2015, p. 21) who likewise claims that most of Elixir’s standard library is implemented as macros — a compile-time abstraction used in metaprogramming in Elixir.

One of the ways to extend the underlying programming language is by creating a domain-specific language on top of it. In fact, Iyengar (2023, p. 151, see description of the example snippet) notes creating a domain-specific language such as the one introduced by the Phoenix Framework ³ in Elixir is impossible without using metaprogramming in Elixir.

To understand metaprogramming, we need to first understand the lifecycle of a program in Elixir. Compile-time is the stage where source code is converted into binary or other format for a machine or a virtual machine for execution. Conversely, run-time ⁴ is referred to as the stage when the code is actually executed (Woo 2021).



Figure 4. Program’s lifecycle according to Woo (2021).

Simplifying, an Elixir source code file is compiled at the compile-time stage first, ultimately into a binary format for the BEAM runtime. During this step Elixir’s code compiler builds an intermediate representation of the source code, representing it as what is called an abstract syntax tree, before it is passed on in the program’s lifecycle, as visualized by Figure 4 above. In Elixir, metaprogramming targets this step as a way to programmatically alter the logic that gets ultimately compiled into Erlang bytecode for execution at run-time (The Elixir Team 2023, see *Adding Elixir to existing Erlang programs* in *Crash Course* section). Test suites are run against the outputs of the program executed at run-time. In mutation testing, the program executed at run-time yields different outputs with mutations than without them, notwithstanding equivalent mutants. Therefore, tests that are able to detect the unexpected outputs, killing the mutants in mutation testing terminology, do that at the run-time step.

5.3 AST and its manipulation

Abstract syntax trees (henceforth AST) are a common concept in compiler engineering. ASTs are sometimes called just syntax trees, or as Cooper and Torczon (2012, p. 227) notes, parser trees due to their close correspondence with them. An AST is a textual representation

3. <https://www.phoenixframework.org/>

4. Run-time is written with a hyphen to make a distinction with a runtime system.

of a syntactic structure that takes the form of a tree (Iyengar 2023, p. 155). As McCord (2015, p. 2) describes, AST is typically an intermediate step when compilers code compilers or interpreters reason about the code before it is turned into bytecode or machine code for run-time execution by the runtime.

In the context of the Elixir programming language, Iyengar (2023, p. 155) calls AST a *tree representation of the structure of source code written in a programming language*. In practice, as Iyengar expands, an AST in Elixir does not encompass all of the details in the source code, just the structure that would impact the execution of the code.

In Elixir, at a high level the AST is generated by a two step process. First, a lexical analysis performed on the source code. Lexical analysis is also called tokenization and it is a process that gets lists of tokens, such as functions, operators and so on from the source code in Elixir. Lexical analysis is followed by a syntactical analysis which is also called parsing. In syntactical analysis the tokens collected with lexical analysis are added to a tree as nodes, which ultimately becomes an AST (Iyengar 2023, p. 155).

For example, consider the following piece of Elixir source code:

```
div(rem(5, 3) - 1, 2)
```

When tokenizing and parsing the above source code, Elixir's compiler would generate an AST that looks structurally like this:

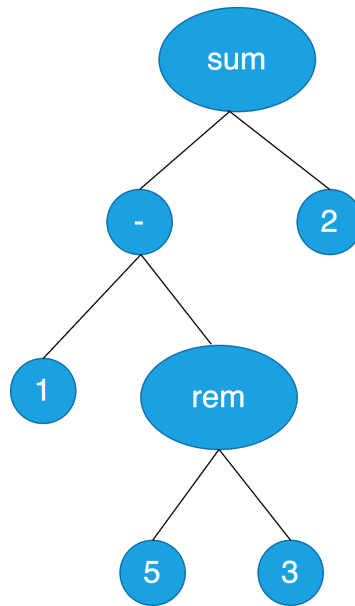


Figure 5. An AST visualization of `div(rem(5, 3) - 1, 2)` Elixir source code.

Elixir represents ASTs in the form of what is referred to as *quoted expressions* (Elixir v1.15.5 2023c, see documentation for `quote/2` in `Kernel.SpecialForms` module). As can be seen from Elixir’s standard library documentation for `quote/2`, quoted expressions are composed of three-element tuples. For example, taking the fragment `rem(5, 3)` from the previous example, the child node in the AST looks like the following:

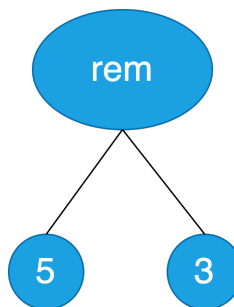


Figure 6. AST representation of the fragment `rem(5, 3)`.

Now, turning this fragment into a quoted expression would produce the following kind of Elixir data structure:

```
{:rem, [context: Elixir, imports: [{2, Kernel}]], [5, 3]}
```

In the three-element tuples of quoted expressions such as the one above:

- The first element is an atom or a tuple in the same three-element representation format.
- The second element represents metadata.
- The third element has the arguments to the function call.

In programming, metadata is auxiliary data that is often used to describe, annotate or otherwise attach information to a data structure (Dictionary.com 2023). Although the name metadata resembles metaprogramming, they are unrelated conceptually — metadata is a common concept in all programming whereas metaprogramming more specific to a subset of programming languages. In Elixir, metadata is part of every AST node that is composed of the three-element tuple previously described.

Going back to the previous Elixir expression example, the first element is the function's name in atom format that is being called, in this case `:rem`. The second metadata element details the context of the expression as well as what modules are imported. Finally, the third element simply lists the two arguments `5` and `3` that were given as arguments to `rem/2`. Armed with an understanding of how Elixir represents ASTs internally as quoted expressions, we are now able to produce the entire quoted expression Elixir's compiler would generate from the expression `div(rem(5, 3) - 1, 2)` as visualized in Figure 5:

```
{:div, [context: Elixir, imports: [{2, Kernel}]],  
 [br/>   {:-,  
    [context: Elixir, imports: [{1, Kernel}, {2, Kernel}]],  
    [{:rem, [context: Elixir, imports: [{2, Kernel}]],  
      [5, 3]}, 1]},  
   2  
 ]}
```

Notably changed from the quoted expression of the fragment `rem(5, 3)`, the last element of the tuple has a structurally different format. Since the last element represents the arguments given to the function call, it now has a quoted expression as the first argument passed to `div/2`. That quoted expression is our fragment `rem(5, 3)`.

The primary means of doing metaprogramming in Elixir is by defining what are called *macros*. Macros are compile-time constructs that get called with Elixir’s quoted expressions as an input and produce quoted expressions as an output. As McCord (2015, p. 4) concisely puts it, macros are essentially *code that writes code*. In practice, this macro system in Elixir allows the programmer to define expressions that transform the AST that is ultimately used at run-time. `Macro` module in Elixir’s standard library provides a set of tooling as helper functions to help with traversing and transforming ASTs (Elixir v1.15.5 2023c). In the construct in this thesis, while macros expressions are not leveraged directly, the `Macro` module’s helpers are used to traverse the AST in search for operators to mutate.

5.4 Testing in Elixir

Software test suites in Elixir are often written by using Elixir’s native testing framework called ExUnit (Elixir School 2023, see Testing). As a part of the standard library of Elixir, it can be used by all projects built on Elixir and is therefore very common in Elixir codebases (Elixir v1.15.5 2023a, note how ExUnit’s source code is a part of Elixir’s own codebase). The construct in this research is tailored for Elixir’s ExUnit testing framework alone and there are no plans to introduce support to any other Elixir testing tool or framework.

As can be glanced from the documentation, ExUnit test suites are most commonly set up by using macros, namely ExUnit’s custom defined `use ExUnit.Case` macro which imports the functionality from ExUnit needed to compose tests and assert on results (Elixir v1.15.5 2023a, see `ExUnit.Case` module for reference). By convention, as is shown in ExUnit’s documentation, ExUnit test suites reside in files that:

- Are named with a appending `_test` in the filename.
- Carry an extension `.exs` which denotes an Elixir script, as opposed to source modules that use an `.ex` extension.

Benefiting from this convention, it is possible to build the construct in this thesis to discover all test modules that are inside an application path. While the construct in this thesis is not built to analyze or anyhow introspect any of the tests written for ExUnit, by traversing the AST generated by compiling the test suite, it would be possible to identify which source

modules each test suite imports directly. This would allow testing injected mutations primarily against test suites that are known to test them directly. This is addressed, along with its inherent drawbacks, in section 7.1 as a potential follow-up improvement method to improve performance of the construct.

6 Mutix

To answer the research questions outlined in chapter 1, a practical mutation testing tool is constructed as a part of this thesis. The tool is implemented for the Elixir programming language and is an open source library called *Mutix*. The source code of Mutix can be found on GitHub ¹. At the time of writing this thesis it is at version `v0.1.1` ² and all the functionality and code referenced in this thesis is a part of this version. The current version of the tool is tested with Elixir's version `v1.15.5` and Erlang/OTP version `26.0.2` ³. All the references in this thesis to Elixir's semantics or compiler internals reference these versions.

Mutix implements mutation testing on any Elixir source file that has a coverage with an ExUnit test suite. As a testing tool, Mutix comes with the following features:

- Injects mutations into a target source file.
- Detects when no target operator present in the source file.
- As the result, generates a mutation report, including mutation score.
- Details undetected mutants line-by-line in the source file.

From the perspective of a developer using this tool, Mutix takes a target source file as an input, injects mutations into it and runs the test existing test suite against them. The ultimate result is a mutation report, which in Mutix encompasses all the detailed feedback the testing tool produces, not just a simple mutation score.

6.1 Mutation operators

As described in the section 3.1, mutation operators are a set of predetermined rules to apply when injecting mutants. Mutix is currently restricted to the following mutation operators: `+`, `-`, `*`, `/`, `and`, `not`, `&&`, `||`, `>`, `>=`, `<`, `<=`, `==` and `!=`. However, instead of applying the mutation operators based on a randomly generated seed number like many of

1. <https://github.com/tuomohopia/mutix>

2. Version tagged with a corresponding git tag of the same name.

3. Elixir requires Erlang/OTP installed to work.

the popular libraries like Stryker Team (2023a, see *configuration options* for mutator) does, Mutix allows the developer to configure these operators per source file to run.

In Mutix's verbiage in the documentation on GitHub, the predetermined rules called mutation operators are defined in terms of *from* and *to* operators where *from* is the target operator found in the source code and *to* the operator to which it is mutated. Any of these operators can be flexibly used as the *from* or *to* operator, allowing the developer to customize what Elixir programming language operators they want to mutate into what, only limited by the currently allowed list of mutation operators. This customization is done in practice by running Mutix with different command line arguments. For example, the following command runs Mutix against a source file found in path `lib/parser.ex`, mutating `+` operators into `-`.

```
mix mutate lib/parser.ex --from + --to -
```

If the `--from` and `--to` command line options are not specified, by default Mutix assigns *from* as `+` and *to* as `-`. With little modifications to the source code of Mutix, the list of allowed operators could be vastly expanded, even to encompass not only arithmetic and other operators defined by the Elixir language but all function calls, whether local or remote. This is discussed briefly as a follow-up improvement in section 6.6.

6.2 Launching Mutix

Mutix is run by invoking a custom written *mix task*⁴. In practice, Mutix is invoked simply by running:

```
mix mutate lib/parser.ex
```

where `lib/parser.ex` is the relative or full path to the source file to mutate. This starts the custom written mix task for Mutix⁵ that encompasses everything required in the entire mutation testing process for Mutix. Initially, after extracting the source file and mutation operator configuration from the command line input, Mutix performs a series of checks to validate that:

4. Custom action defined by Elixir's integrated build tool (Elixir v1.15.5 2023c).

5. <https://github.com/tuomohopia/mutix/blob/v0.1.1/lib/mix/mutate.ex#L63-L186>

- The defined mutation operators are within the allowed operators.
- Only one source file path is supplied.
- The source file exists on the supplied path.
- The source file belongs to a valid Mix project.

After the initial checks Mutix proceeds to using Elixir's compiler to compile the codebase by programmatically invoking the code compilation Mix task:

```
Mix.Task.run("compile", [])
```

As documented in Elixir's source code documentation, running this Mix task uses the compilers defined in the Mix project to compile the code for the application, which by default are defined as `[:yecc, :leex, :erlang, :elixir, :app]` (Elixir v1.15.5 2023b, see *mix compile*).

Importantly, after the code for the application has been compiled Mutix *unrequires* the source file by using Elixir's standard library's `Code.unrequire_files/1` function:

```
Code.unrequire_files([source_file])
```

Although Elixir's documentation on this is thin, unrequiring the source file removes the source module to mutate from a list of modules to track in terms of compilation. Effectively, this allows Mutix to trigger compilation for new, mutated versions of the source file as a part of the application as they are generated by Mutix from the source code.

To finish up the initial procedures Mutix finds and compiles all the test files in the application's test suite, filtering out those that do not contain any ExUnit test suites. After the ExUnit test suites of the application have been located, Mutix uses Elixir's compiler to compile all the test files concurrently:

```
Kernel.ParallelCompiler.require(test_files, [])
```

Finally, Mutix runs the standard test suite, capturing and muting its printed feedback, without any mutations injected in order to assert that the test suite runs with no failures against the original source code. Sometimes this is called a *dry* run and is not unlike how many other

mutation testing libraries work ⁶.

6.3 Injecting mutations

As soon as Mutix has performed its initial checks and compiled the application's original source code, along with the full test suite, reads the source code in binary format from the file it finds on the file system. With the aid of Elixir's compiler helper functions, `Code.string_to_quoted!/1` is invoked with the source file contents in order to produce an AST — an Elixir representation of the source code. The generated AST gets passed to Mutix's AST transformer function whose role is to:

1. Locate all occurrences of the given *from* operator in the source code.
2. Generate a transformed AST with a single mutation injected for each occurrence.

Since the locating of the *from* operators in the source code happens before any source code is executed, this step is essentially a form of static analysis, as introduced in section 2.1. For the subsequent step, the diagram below illustrates how Mutix's AST transformer works at a high level:

6. For example: <https://github.com/stryker-mutator/stryker-js/blob/master/packages/core/src/stryker.ts#L40>

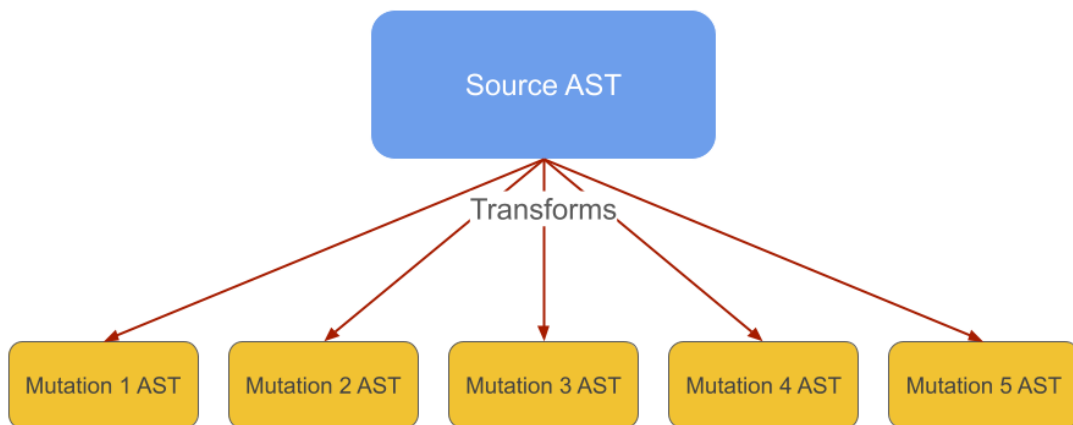


Figure 7. Source AST transformed into mutated ASTs.

Essentially, the original, unaltered source AST is transformed into a list of new ASTs, each containing the source AST but with a single mutation injected. In the above illustration, Mutix found five occurrences of the *from* operator and generated a corresponding five *to* mutations from them by means of AST transformations. Ultimately, as illustrated, this manifests as five entirely new ASTs, each with a single mutation in them. Although in the current version the transformed ASTs are generated sequentially, parallelizing the operation would be fairly trivial as the transformations happen independently of each other, just from the same source code.

The following example paints a more practical picture of how Mutix's transformer works. Given the following Elixir module with two comparison functions, one for numbers and one for timestamps:

```
defmodule Comparer do
  def greater?(a, b) do
    a > b
  end
end
```

```

def later?(timestamp1, timestamp2) do
  timestamp1 = DateTime.to_unix(timestamp1)
  timestamp2 = DateTime.to_unix(timestamp2)
  timestamp1 > timestamp2
end
end

```

Mutix would initially transform this source module into an Elixir AST structure where the `greater?/2` and `later?/2` functions would be represented by the following AST fragments:

```

# greater?/2:
{:def, [line: 2],
 [
   {:greater?, [line: 2],
    [
      [
        {:a, [line: 2], nil},
        {:b, [line: 2], nil}
      ]},
      [do: {>, [line: 3],
        [
          [
            {:a, [line: 3], nil},
            {:b, [line: 3], nil}
          ]}
        ]}
    ]}
 ]},

```

```

# later?/2
{:def, [line: 6],
 [
   {:later?, [line: 6],
    [
      [
        {:timestamp1, [line: 6], nil},
        {:timestamp2, [line: 6], nil}
      ]},
      [
        do: {__block__, [],
          [
            [
              {:=, [line: 7],
                [

```

```

{:timestamp1, [line: 7], nil},
{:.., [line: 7],
[{:__aliases__, [line: 7], [:DateTime]}, :to_unix]},
[line: 7], [{:timestamp1, [line: 7], nil}}}
]},
{:=, [line: 8],
[
{:timestamp2, [line: 8], nil},
{:.., [line: 8],
[{:__aliases__, [line: 8], [:DateTime]}, :to_unix]},
[line: 8], [{:timestamp2, [line: 8], nil}}}
]},
{:>, [line: 9],
[{:timestamp1, [line: 9], nil},
{:timestamp2, [line: 9], nil}}}
]}
]
]}

```

Now, if Mutix was run with *from* operator defined as `>` and *to* operator as `<`, Mutix would initially be looking for the `>` occurrences in the source AST. Although the AST looks noisy when printed out, we can easily identify the two different occurrences there are for the operator `>` which are represented as atoms⁷ in the AST:

```

# inside greater?/2
{:>, [line: 3], [{:a, [line: 3], nil},
{:b, [line: 3], nil}]}

# inside later?/2
{:>, [line: 9],
[{:timestamp1, [line: 9], nil},

```

7. Constants whose value is their name. Syntactically represented with a leading `:`.

```
{:timestamp2, [line: 9], nil}}}
```

The fragments above are nodes of the AST where the operator `>` is found. Technically, Mutix finds⁸ these occurrences by traversing the AST using Elixir’s *Macro* module’s helper function, namely `Macro.prewalk/3` which is a pre-order, depth-first traversal function built for quoted expressions, in other words Elixir ASTs (Elixir v1.15.5 2023a, see `Macro` module). Conveniently, much like folds typically found in functional languages, `Macro.prewalk/3` allows aggregating an accumulator throughout the traversal, allowing Mutix to pack metadata to it. For each iteration, the node tuple’s first element is pattern matched against the *from* operator, in this case against `:>` which is an atom form of the operator. If a match is found, Mutix will add details about the occurrence to the metadata accumulated throughout the traversal.

Once the entire AST has been traversed and all occurrences of the *from* operator found, Mutix will start producing transformations⁹ of the AST where one single mutation — *from* is mutated to *to* — is injected in each produced AST. This transformation is likewise implemented with a pre-order, depth-first traversal function. But instead of just injecting the mutation to the target node, Mutix also updates the said node’s metadata to detail the index of the occurrence on that given line of code. This allows Mutix’s report producer to highlight the operator in the source code when producing feedback to the developer on surviving mutants.

Ultimately, what Mutix’s transformer returns is a list of new ASTs along with some helpful metadata for reporting purposes. The number of ASTs transformed matches the number of mutants generated as only a single mutant is injected per AST.

6.4 Report

Having finished the mutation injections, Mutix will, for each AST with a mutation injected:

1. Compile the AST with a mutation injected as a part of the application.
2. Run the entire ExUnit test suite captured and compiled by Mutix earlier, aggregating results.

8. <https://github.com/tuomohopia/mutix/blob/v0.1.1/lib/mutix/transform.ex#L17-L26>

9. <https://github.com/tuomohopia/mutix/blob/v0.1.1/lib/mutix/transform.ex#L38-L44>

Building on Figure 7, at a high level the mutation testing process in Mutix would continue like this:

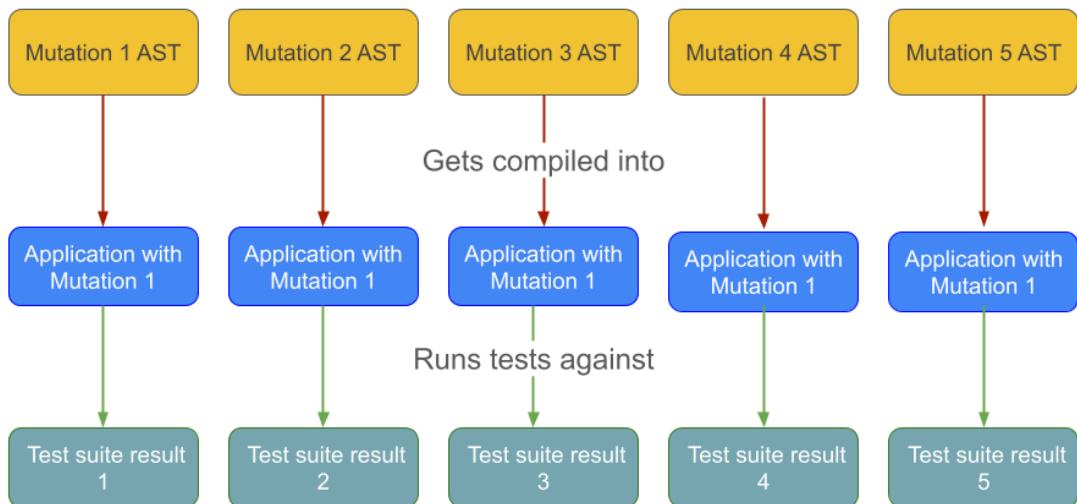


Figure 8. Test suite is run against each version of the application with a single mutant injected.

Aside from printed feedback, each ExUnit run programmatically produces the following kind of example Elixir data structure as a result (Elixir v1.15.5 2023a, refer to `suite_result()` data type):

```
%{
  failures: 1,
  total: 5,
  excluded: 0,
  skipped: 0
}
```

Since the entire ExUnit test suite is run against each mutated version of the application, Mutix will generate the same amount of the above structures as there are mutations injected. With each test run, Mutix will receive the above ExUnit result data structure and zip it to-

gether with metadata about the injected mutation's location in the source code. This allows Mutix reporter to aggregate feedback of the step where Mutix determines if the test suite is able to capture and kill each mutant or not. Visualized, this step looks like the following:

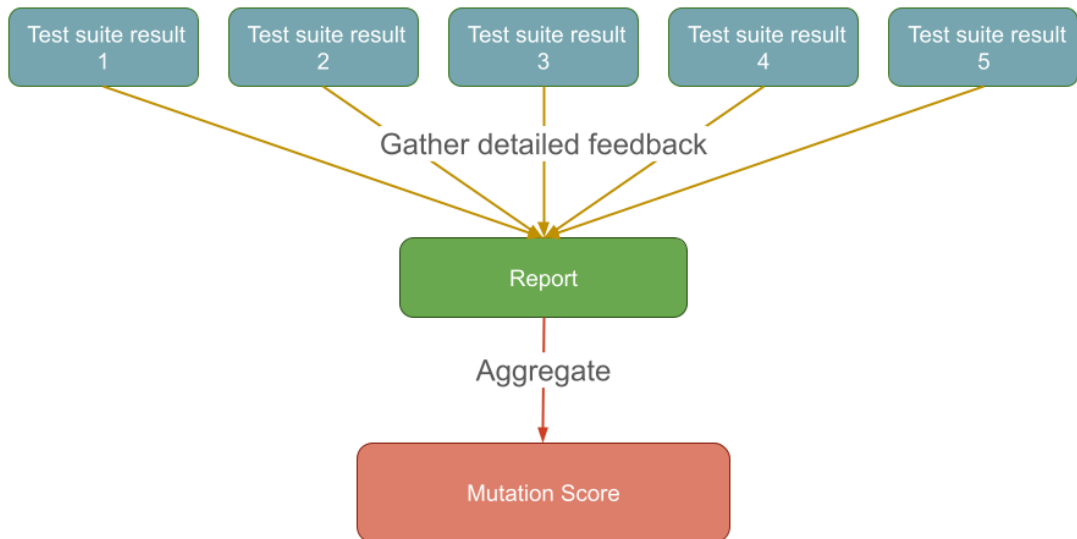


Figure 9. Test suite is run against each version of the application with a single mutant injected and results aggregated into a mutation report and ultimately, a mutation score.

A test suite for the earlier defined `Comparer` source module in the code block in 6.3 might look something like this:

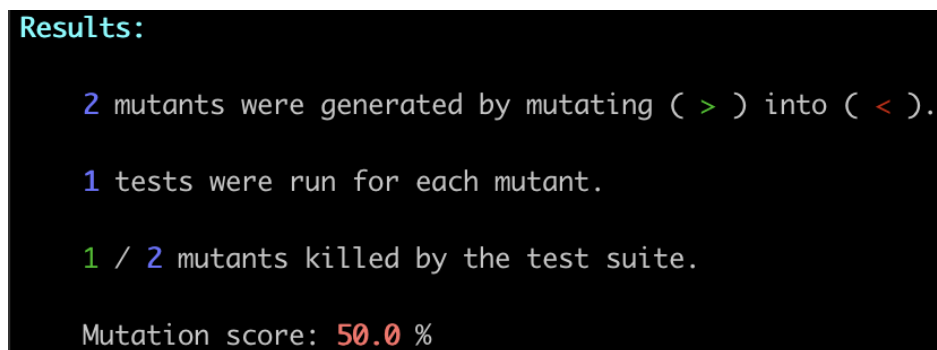
```
defmodule ComparerTest do
  use ExUnit.Case

  describe "greater?/2" do
    test "confirms the 1st number is greater than the 2nd" do
      assert Comparer.greater?(2, 1) == true
    end
  end
end
```

With this test suite having all the tests in the application, the application's two functions now have test coverage for only one of them. To run Mutix injecting `<` in place of all occurrences of `>` the following command is used:

```
mix mutate lib/comparer.ex --from ">" --to "<"
```

This will run the test suite with Mutix, having two mutations injected to the source module. Therefore, the test suite will be run twice — once for each mutation. As a result, Mutix will print out the following aggregate of the test runs:



```
Results:  
  
2 mutants were generated by mutating ( > ) into ( < ).  
  
1 tests were run for each mutant.  
  
1 / 2 mutants killed by the test suite.  
  
Mutation score: 50.0 %
```

Figure 10. Mutation report along with mutation score.

As can be seen from the feedback Mutix produces, the report details the following results:

- The number of mutants injected and which mutation operator was used.
- How many tests were run per mutant.
- How many mutants survived from the total mutants injected.
- Mutation score.

As defined in section 3.1 a surviving mutant is an infection that no test is able to capture by failing. In this case, one mutant managed to evade capture by the test suite as the infection had no test coverage with the defined software test suite. As the report about surviving mutants, Mutix will print out the following kind of feedback:

```
Surviving mutants - no test failed with these injections:  
  
lib/comparer.ex:9 where ( > ) was mutated into ( < ):  
  
    timestamp2 = DateTime.to_unix(timestamp2)  
    timestamp1 > timestamp2  
end
```

Figure 11. Mutix's feedback for surviving mutants.

Technically, Mutix's reporter receives the ExUnit test suite run result data structure along with metadata, detailing the infection itself. With the metadata about the injected mutant that was produced during the AST transformation when injecting mutations, a helper function ¹⁰ in the reporter is able to produce a report detailing the:

- The mutation operator used.
- The source file path and line where the surviving mutant is located.
- Context in the source file with highlight on the actual surviving mutant on the line.

This feedback gets printed for every mutant that has survived running the test suite with Mutix. Finally, as can be seen in Figure 10 Mutix computes a mutation score ¹¹ of the test suite run with Mutix. Revisiting the mutation score defined in section 4.1, mutation score is a percentage of the killed mutants divided by the number of all infections which is subtracted by the count of equivalent mutants. As Mutix, just like most other mutation testing tools, fails to detect equivalent mutants that do not alter the semantics of the program, mutation score is simply the amount of killed mutants from total mutants injected. In this case, as the test suite has coverage for only a single mutant, one from two mutants is captured while the other one survives, resulting in a 50% mutation score.

10. <https://github.com/tuomohopia/mutix/blob/v0.1.1/lib/mutix/report.ex#L70-L110>

11. <https://github.com/tuomohopia/mutix/blob/v0.1.1/lib/mutix/report.ex#L51-L68>

6.5 Performance

To get a rough idea of the performance impact Mutix comes with, a naive example program with three functions, each containing a single `+` operator in a single source file was constructed ¹². The performance assessment was conducted at the example program's version denoted by git hash `6e72cef`, using the same Elixir and Erlang/OTP versions as for constructing Mutix itself. The performance of Mutix was assessed simply by measuring the execution time of running Mutix, comparing to the execution time of the example program's ExUnit test suite itself. The example program's test suite had to be slightly modified to run the test suite thrice to match the number of ExUnit test suite runs Mutix does as it produces three mutations of the source program, running the test suite once against each one of them. This was done by creating two additional, identical copies of the test suite, just with a different module name.

Before running the standard ExUnit test suite the code was precompiled, so that compilation itself would not alter the execution results:

```
mix compile
```

Following the successful compilation of the source code, the test suite was run, clocking the time with Linux's `time` utility:

```
time mix test
```

This command was run a thousand times and the average execution time measured for the test suite was 0.45 seconds. Subsequently, Mutix was run on the same source program and test suite with the same `time` utility:

```
time mix mutate lib/mutater.ex --from + --to -
```

This was likewise repeated a thousand times to get the average execution time, which yielded 0.48 seconds. Each time Mutix is invoked against a source file, it runs the ExUnit test suite itself against each mutated version of the source program. Subtracting the average test suite execution time of 0.45 seconds from Mutix's average execution time of 0.48 seconds leaves

12. <https://github.com/tuomohopia/mutater>

an average 0.03 second execution time for Mutix’s own logic per execution — a fraction of the test execution time itself. Each Mutix execution produced the correct mutation score and feedback from running the tool.

Most of the work Mutix does on top of running the ExUnit test suite are the AST transformations to produce the mutated source modules. This simple execution time measurement would indicate that this is a fairly fast process with Elixir’s compiler that Mutix utilizes. However, while Mutix may not add a lot of performance overhead on top of running the test suite itself, the full test suite gets run as many times as there are mutated versions of the source program. This adds up quickly, making this the largest drawback of the current implementation of Mutix.

For example, if the source program had 10 source modules of similar size, each with the same three operators to mutate, Mutix would produce a total of 30 mutated versions of the source program when running Mutix on each source module. The entire test suite would then get run against each mutated version in order to derive a mutation score and feedback. Using the execution times from this performance assessment, this would result in a total ExUnit test suite execution time of $0.45 * 30 = 13.5$ seconds. While Mutix’s own logic execution would only add 0.03 seconds per source module to the total execution time, thus mere $10 * 0.03 = 0.3$ seconds, the major issue in terms of execution time is the repeated ExUnit full test suite run against each source module. Solutions to this are discussed in section 7.1.

Although this performance assessment was shallow, only measuring execution times with a static example application, it gives some preliminary indication of the performance penalty Mutix’s own logic imposes on top of running the ExUnit test suites. The execution times measured in this assessment would indicate that Mutix adds only very light performance overhead to conventional ExUnit test execution, but introduces the problem of having to run the test suite as many times over as there are mutations injected.

6.6 Caveats

In the current version, a notable caveat is that Mutix does not discriminate operators to mutate based on the context they reside in. Essentially, this means that given a source code file where there is code with the given target operators that is invoked at compile-time, Mutix will indiscriminately also inject mutations there. This is unintentional but known and documented behavior in the current version.

For example, consider the following Elixir code where a module attribute is assigned a value by dynamically computing an expression at compile-time:

```
defmodule SourceModule do
  @incrementor 1 + 1

  def increment(a), do: a + @incrementor
end
```

In the mock source module definition above, `@incrementor` gets its value from the evaluation of the expression `1 + 1` that happens upon compilation. Now, Mutix ingests the entire source code file and finds the target operators to mutate, injecting mutations before the compilation takes place. Therefore, without context aware mutation algorithms this naturally injects mutations into the same target operators everywhere, including code that is run at compile-time instead of run-time. Consequentially, when Mutix is run with `+` as the target operator to mutate, for example with `-` as the mutation operator, it would also mutate the `@incrementor` value that is used at run-time by the `increment` function. With this, `@increment` would evaluate to `1 - 1 = 0` at run-time, essentially transforming the semantics of the `increment` function into:

```
def increment(a), do: a + 0
```

Obviously, transforming the compile-time logic with mutation injections can lead to test results that are hard to reason about. Therefore, at this moment the developer using Mutix will simply have to be aware of this when defining the source code to run Mutix against. Ways to make Mutix context aware in order to beat this issue is discussed in the section 7.1

section.

While not a real caveat, the range of mutation operators Mutix currently allows is slim. Although new operators to the list of allowed operators could be added trivially, implementing local or remote function calls requires building entirely new logic. For example, consider an unmutated line of code with two remote calls to `List` module's functions that adds up the first and last element of a given list:

```
List.first(a) + List.last(a)
```

As an AST node of the remote function call `List.first(a)` would be represented in the following format:

```
{{:., []},  
  [{:__aliases__, [alias: false], [:List]}, :first]}, [],  
  [{:a, [], Elixir}]}
```

Now, as nodes that describe remote function calls simply have a different format AST node with different children nodes than for instance an arithmetic `+` node would have, Mutix's AST transformation algorithms could be extended to work on them with moderate effort. This would allow Mutix to practically inject mutations into any piece of executable code in the source module. However, as this goes generally beyond what typical industrial mutation testing software do, it is questionable how this extension should be designed so that it is viable and sensible from a developer perspective to use to improve software test quality.

Other things that can be considered as caveats in the current implementation are mainly related to performance and user experience. As detailed in the section 6.3, Mutix not only creates injections sequentially but also runs the entire test suite against every single injection, one by one, making running Mutix resource intensive computationally. To keep the user experience bearable, Mutix currently is restricted to one single source module at a time. A practical proposal to alleviate this issue are introduced in section 7.1.

7 Discussion

The main purpose of mutation testing, as stated in chapter 3, is to assert the test quality and coverage of program by means of testing the application against mutated versions of it. Building on this premise, the original research questions in the chapter 1 were to find out if mutation testing can be applied in a functional programming context and with developer friendly feedback produced.

Mutix, as implemented as a part of this thesis, is able to:

- Inject mutations with a range of configurable mutation operators into a source program.
- Assert test quality individually per mutant.
- Produce a mutation score.
- Detail developer friendly feedback on surviving mutants.

With these features, built in a functional programming context, Mutix is considered to have fulfilled the original research goals that were set for the thesis. As per the definition of higher order hierarchies in mutation testing by Harman, Jia, and Langdon (September 2010) Mutix falls into the *First Order Mutation Testing* category. This is because each AST with a single-order mutation is tested against the full test suite for each iteration. To qualify for higher-order mutation testing, among other things the test suite would have to be run against multiple simultaneous mutations.

In the traditional mutation testing process presented by Offutt and Untch, as visualized in Figure 1, maintaining the original test suite so that it is run successfully is also included in the process, not only running the mutation testing tool. Mutix runs the existing ExUnit test suite silently on the original program before proceeding to run the same test suite against the mutated versions of the program. Therefore, Mutix implements the traditional mutation testing process fully, excluding the manual marking of equivalent mutants which is not currently supported.

The technical means of implementing mutant injection and mutation testing of existing industrial tools was not evaluated in this thesis. Exploring the source code of popular, open

source mutation testing solutions such as those listed in Table 1 shows that there appears to be no consistent patterns when it comes to the methods used when it comes to practical implementations of the testing tools. Mutix leverages metaprogramming for this which is introduced in section 5.2. Mutix's implementation with metaprogramming is detailed in section 6.3.

7.1 Limitations & future improvement

Currently, Mutix has no way to reason about or manually ignore equivalent mutants injected to the program. As referenced, earlier while discussing the currently available industrial solutions, most other mutation testing tools are in the same pit and have generally taken the approach to educate the developer about the issue of equivalent mutations in the tool's documentation. Although this is a known limitation in Mutix as well, Mutix's documentation does not yet educate the programmer about the issue with equivalent mutants.

Aside from this weakness, Mutix, at its current version, has poor technical documentation and only supports a fixed amount of predetermined mutation operators. The extension of Mutix to support arbitrary operators, including local and remote function calls is explored in section 6.6. To summarize, aside from thin technical documentation and a suppressed amount of mutation operators, Mutix currently has the following known caveats and limitations:

1. Mutates operators even in code run compile-time.
2. Not performance optimized and as a result, only accepts a single source file at a time.
3. No detection or recognition of equivalent mutants.

Out of these known drawbacks, the first two items are somewhat vital to tackle before Mutix can realistically be used in any industrial setup. This is because respectively:

1. Compile-time evaluated code is often prevalent in larger Elixir codebases.
2. Industrial codebases often have large codebases and test suites which take time to compile and run.

To prevent infecting compile-time code with mutations, the most pertinent solution would

be to simply make the algorithms that locate the *to* operators to mutate context aware. In practice, an easy way to implement this would be to simply have the algorithms detect if the operator to mutate is discovered as a child of a function declaration node in the AST. This is next on the list of improvements to implement for Mutix and requires only a moderate engineering lift to implement — just tweaks to the current AST traversing algorithms.

However, the issue of inadequate performance of mutation testing with Mutix is a more difficult problem to solve. While, Mutix could allow running the tool against the entire codebase at once, generating mutations in every source file where the *from* operator is found, it would result in a potentially large amount of mutants generated and tested against. Although the discovery of operators to mutate as well as injection of mutants is trivial to parallelize computationally, the entire test suite would still have to be run sequentially against each mutated version of the application.

The issue should primarily be approached from the angle of reducing the number of tests to run against each mutated version of the application while still maintaining enough confidence that all the tests that could potentially find and kill the mutation are run. An implicit way to do this that would require no further input from the programmer would be to have Mutix perform a static analysis on the test suite to see which tests alias or import the functions directly where the injected mutant is located. This would most commonly mean that only unit tests that test the functions directly get run since integration or other higher level tests may not invoke the mutated functions directly but indirectly through another module.

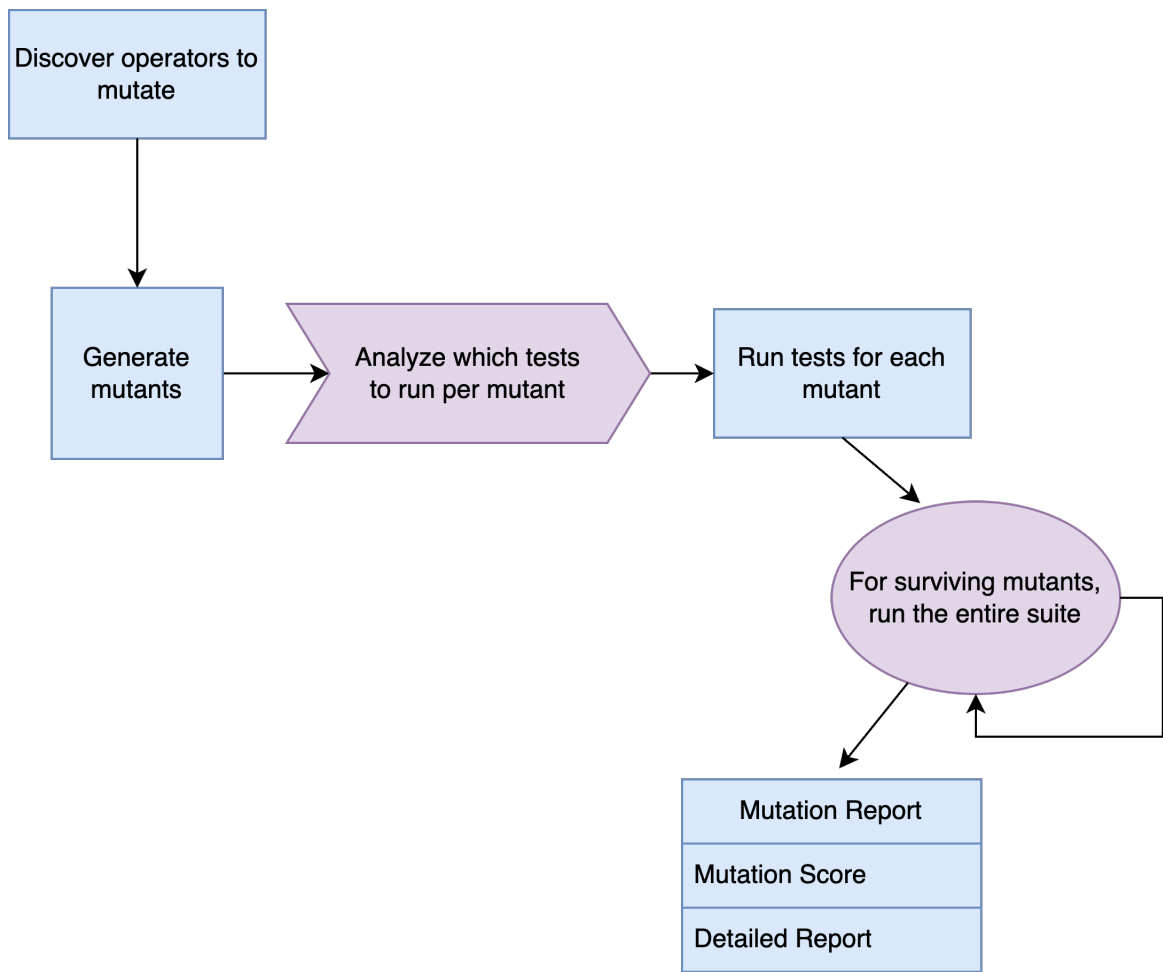


Figure 12. Proposal to improve performance of Mutix. Light blue items denote the current steps while the light purple items display the new steps to introduce.

After the narrowed down list of tests was run on each mutant, Mutix would then proceed to run the entire test suite only for the surviving mutants. This is in order to reaffirm that no test in the codebase actually provide coverage to identify and kill the mutant that had survived the narrowed down list of tests run on it.

Holistically, implementing an upgrade such as the one described above would potentially substantially reduce the number of tests that get run, because often it is the role of unit tests to already provide test coverage for the logic of functions. As a result, this improvement would lead to performance benefits particularly in those codebases the unit test coverage was already extensive to begin with. Codebases with poor test coverage for unit tests that

import or alias the tested code directly would potentially lead to performance regressions as mutations would get partially tested twice — first with targeted unit tests and then again when the full test suite was run for the surviving mutant.

Other projected lighter improvements to Mutix are upgrades to the developer experience. For example, the mutation report could be exported as a HTML document. Moreover, to integrate with continuous integration and delivery pipelines found in industrial software engineering projects, Mutix would also have to allow a configurable mutation score threshold and produce an appropriate Unix error code accordingly.

Compared to many other practical mutation testing software publicly available, Mutix has obviously only taken its baby steps and is not industrially tested as a testing tool. Further improvement steps require exploration and getting some practical user experiences first from developers actually using the tool to systematically assert and improve the quality of their test suites in their software projects.

Bibliography

Ammann, Paul, Marcio Eduardo Delamaro, and Jefferson Offutt. 2014. “Establishing Theoretical Minimal Sets of Mutants”. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 21–30. <https://doi.org/10.1109/ICST.2014.13>.

Ammann, Paul, and Jefferson Offutt. 2017. *Introduction to Software Testing*. 2nd edition. Cambridge University Press. <https://doi.org/10.1017/CBO9780511809163>.

Andrews, J.H., L.C. Briand, and Y. Labiche. 2005. “Is mutation an appropriate tool for testing experiments? [software testing]”. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. 402–411. <https://doi.org/10.1109/ICSE.2005.1553583>.

Armstrong, Joe. 2007. “A History of Erlang”. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, 6-1-6–26. HOPL III. San Diego, California: Association for Computing Machinery. ISBN: 9781595937667. <https://doi.org/10.1145/1238844.1238850>.

Bourque, Pierre, Richard E. Fairley, and IEEE Computer Society. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd edition. Washington, DC, USA: IEEE Computer Society Press. ISBN: 0769551661.

Breiman, L. October 2001. “Random Forests”. *Machine Learning* 45 (): 5–32. <https://doi.org/10.1023/A:1010950718922>.

Coles, Henry. 2023. “Snippet of PITtest coverage report of Wicket Core test suite”. Visited on May 10, 2023. <https://pittest.org/>.

Coles, Henry, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. July 2016. “PIT: a practical mutation testing tool for Java (demo)”, 449–452. <https://doi.org/10.1145/2931037.2948707>.

Cooper, Keith D., and Linda Torczon. 2012. “Chapter 4 - Context-Sensitive Analysis”. In *Engineering a Compiler*, 2nd edition, edited by Keith D. Cooper and Linda Torczon, 161–219. Boston: Morgan Kaufmann. ISBN: 978-0-12-088478-0. <https://doi.org/10.1016/B978-0-12-088478-0.00004-9>.

Coralogix. 2015. “This is what your developers are doing 75 percent of the time, and this is the cost you pay”. Visited on May 9, 2023. <https://coralogix.com/blog/this-is-what-your-developers-are-doing-75-of-the-time-and-this-is-the-cost-you-pay/>.

Demillo, Richard, R.J. Lipton, and Fred Sayward. May 1978. “Hints on Test Data Selection: Help for the Practicing Programmer”. *Computer* 11 (0): 34–41. <https://doi.org/10.1109/C-M.1978.218136>.

Dictionary.com. 2023. “Metadata”. Visited on November 27, 2023. <https://www.dictionary.com/browse/metadata>.

Domkin, Vsevolod. 2021. *Programming Algorithms in Lisp*. 1–377. Apress Berkeley, CA. <https://doi.org/10.1007/978-1-4842-6428-7>.

Elixir School. 2023. “Elixir School - metaprogramming”. Visited on November 19, 2023. <https://elixirschool.com/en/lessons/advanced/metaprogramming>.

Elixir v1.15.5. 2023a. “ExUnit”. Visited on June 8, 2023. https://hexdocs.pm/ex_unit/1.15.5/ExUnit.html.

———. 2023b. “Mix”. Visited on November 21, 2023. <https://hexdocs.pm/mix/1.15.5/Mix.html>.

———. 2023c. “Standard Library”. Visited on November 20, 2023. <https://hexdocs.pm/mix/1.15.5>.

Falah, Bouchaib, Bouriat Salwa, and Ouidad Achahbar. September 2013. “Effectiveness of Mutation Testing Techniques: Reducing Mutation Cost”. In *2013 World Congress on Multimedia and Computer Science (ACEEE 2013)*. Hammamet, Tunisia.

Floyd, Robert W. 1993. “Assigning Meanings to Programs”. In *Program Verification: Fundamental Issues in Computer Science*, edited by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, 65–81. Dordrecht: Springer Netherlands. ISBN: 978-94-011-1793-7. https://doi.org/10.1007/978-94-011-1793-7_4.

Frankl, Phyllis G., Stewart N. Weiss, and Cang Hu. 1997. “All-uses vs mutation testing: An experimental comparison of effectiveness”. *Journal of Systems and Software* 38 (3): 235–253. ISSN: 0164-1212. [https://doi.org/10.1016/S0164-1212\(96\)00154-9](https://doi.org/10.1016/S0164-1212(96)00154-9).

GitHub, Inc. 2022. “Octoverse The state of open source software - Developers”. Visited on May 7, 2023. <https://octoverse.github.com/2022/developer-community>.

———. 2023. “GitHub Search”. Visited on May 15, 2023. <https://github.com/>.

Gopinath, Rahul, Carlos Jensen, and Alex Groce. 2014. “Mutant census: an empirical examination of the competent programmer hypothesis”. Harvard Dataverse. <https://doi.org/10.7910/DVN/25443>.

Hałas, Konrad. 2023. “MutPy Documentation”. Visited on May 15, 2023. <https://github.com/mutpy/mutpy>.

Harman, Mark, and Yue Jia. September 2011. “An Analysis and Survey of the Development of Mutation Testing”. *IEEE Transactions on Software Engineering* 37 (): 649–678. <https://doi.org/10.1109/TSE.2010.62>.

Harman, Mark, Yue Jia, and William Langdon. September 2010. “A Manifesto for Higher Order Mutation Testing”. In *ICSTW 2010 - 3rd International Conference on Software Testing, Verification, and Validation Workshops*, 80–89. <https://doi.org/10.1109/ICSTW.2010.13>.

How Tai Wah, K.S. 2001a. “Theoretical Insights into the Coupling Effect”. Edited by W. Eric Wong. (Boston, MA), 62–70. https://doi.org/10.1007/978-1-4757-5939-6_11.

———. 2001b. “Theoretical Insights into the Coupling Effect”. In *Mutation Testing for the New Century*, edited by W. Eric Wong, 62–70. Boston, MA: Springer US. ISBN: 978-1-4757-5939-6. https://doi.org/10.1007/978-1-4757-5939-6_11.

———. 2003. “An analysis of the coupling effect I: single test data”. *Science of Computer Programming* 48 (2): 119–161. ISSN: 0167-6423. [https://doi.org/10.1016/S0167-6423\(03\)00022-4](https://doi.org/10.1016/S0167-6423(03)00022-4).

Howden, William. March 1978. “Algebraic Program Testing”. *Acta Informatica* (Berlin, Heidelberg) 10, number 1 (): 53–66. ISSN: 0001-5903. <https://doi.org/10.1007/BF00260923>.

———. 1982. “Weak Mutation Testing and Completeness of Test Sets”. *IEEE Transactions on Software Engineering* SE-8 (4): 371–379. <https://doi.org/10.1109/TSE.1982.235571>.

Hudak, Paul. September 1989. “Conception, Evolution, and Application of Functional Programming Languages”. (New York, NY, USA) 21, number 3 (): 359–411. ISSN: 0360-0300. <https://doi.org/10.1145/72551.72554>.

“IEEE Standard Glossary of Software Engineering Terminology”. 1990. *IEEE Std 610.12-1990*, 1–84. <https://doi.org/10.1109/IEEESTD.1990.101064>.

Iyengar, Aditya. 2023. *Build Your Own Web Framework in Elixir: Develop lightning-fast web applications using Phoenix and metaprogramming*. 151–183. Packt Publishing Ltd.

Jia, Yue, and Mark Harman. 2008. “MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language”. In *Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)*, 94–98. <https://doi.org/10.1109/TAIC-PART.2008.18>.

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. “Are Mutants a Valid Substitute for Real Faults in Software Testing?” In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 654–665. FSE 2014. Hong Kong, China: Association for Computing Machinery. ISBN: 9781450330565. <https://doi.org/10.1145/2635868.2635929>.

Kim, Mingwan, Neunghoe Kim, and Hoh In. August 2020. “Investigating the Relationship Between Mutants and Real Faults with Respect to Mutated Code”. *International Journal of Software Engineering and Knowledge Engineering* 30 (): 1119–1137. <https://doi.org/10.1142/S021819402050028X>.

Kintis, Marinos, Mike Papadakis, and Nicos Malevris. April 2014. “Employing second-order mutation for isolating first-order equivalent mutants”. *Software Testing, Verification and Reliability* 25 (). ISSN: 1099-1689. <https://doi.org/10.1002/stvr.1529>.

Langdon, William B., Mark Harman, and Yue Jia. 2010. “Efficient multi-objective higher order mutation testing with genetic programming”. *Journal of Systems and Software* 83 (12): 2416–2430. ISSN: 0164-1212. <https://doi.org/10.1016/j.jss.2010.07.027>.

Laurent, T., S. Gaffney, and A. Ventresque. April 2022. “Re-visiting the coupling between mutants and real faults with Defects4J 2.0”. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 189–198. Los Alamitos, CA, USA: IEEE Computer Society. <https://doi.org/10.1109/ICSTW55395.2022.00042>.

Madeyski, Lech, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. 2014. “Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation”. *IEEE Transactions on Software Engineering* 40 (1): 23–42. <https://doi.org/10.1109/TSE.2013.44>.

McCord, Chris. 2015. *Metaprogramming Elixir: Write Less Code, Get More Done (and Have Fun!)* 1st edition. Pragmatic Bookshelf. ISBN: 1680500414.

Morell, L.J. 1988. “Theoretical insights into fault-based testing”. In *1988 Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, 45062–. <https://doi.org/10.1109/WST.1988.5353>.

Niedermayr, Rainer, Elmar Juergens, and Stefan Wagner. 2016. “Will My Tests Tell Me If I Break This Code?” In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 23–29. CSED ’16. Austin, Texas: Association for Computing Machinery. ISBN: 9781450341578. <https://doi.org/10.1145/2896941.2896944>.

Offutt, Jefferson. 1989. “The Coupling Effect: Fact or Fiction”. In *Proceedings of the ACM SIGSOFT ’89 Third Symposium on Software Testing, Analysis, and Verification*, 131–140. TAV3. Key West, Florida, USA: Association for Computing Machinery. ISBN: 0897913426. <https://doi.org/10.1145/75308.75324>.

———. January 1992. “Investigations of the Software Testing Coupling Effect”. *ACM Transactions on Software Engineering and Methodology* (New York, NY, USA) 1, number 1 (): 5–20. ISSN: 1049-331X. <https://doi.org/10.1145/125489.125473>.

Offutt, Jefferson, and J. Huffman Hayes. 1996. “A Semantic Model of Program Faults”. (San Diego, California, USA), ISSTA ’96, 195–200. <https://doi.org/10.1145/229000.226317>. <https://doi.org/10.1145/229000.226317>.

- Offutt, Jefferson, and Stephen Lee. 1991. "How Strong is Weak Mutation?" In *Proceedings of the Symposium on Testing, Analysis, and Verification*, 200–213. TAV4. Victoria, British Columbia, Canada: Association for Computing Machinery. ISBN: 089791449X. <https://doi.org/10.1145/120807.120826>.
- Offutt, Jefferson, and Roland Untch. 2001. "Mutation 2000: Uniting the Orthogonal". In *Mutation Testing for the New Century*, 34–44. USA: Kluwer Academic Publishers. ISBN: 0792373235.
- Papadakis, Mike, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. "Threats to the Validity of Mutation-Based Test Assessment". In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 354–365. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery. ISBN: 9781450343909. <https://doi.org/10.1145/2931037.2931040>.
- Papadakis, Mike, Yue Jia, Mark Harman, and Yves Le Traon. 2015. "Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique". In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 1:936–946. <https://doi.org/10.1109/ICSE.2015.103>.
- Papadakis, Mike, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. "Chapter Six - Mutation Testing Advances: An Analysis and Survey", edited by Atif M. Memon, 112:275–378. *Advances in Computers*. Elsevier. <https://doi.org/10.1016/bs.adcom.2018.03.015>.
- Papadakis, Mike, and Nicos Malevris. April 2010. "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies". In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 90–99. <https://doi.org/10.1109/ICSTW.2010.50>.
- Papadakis, Mike, Nicos Malevris, and Marinos Kintis. January 2010. "Mutation Testing Strategies - A Collateral Approach". In *ICSOF 2010 - Proceedings of the 5th International Conference on Software and Data Technologies*, 2:325–328.
- Parsai, Ali, and Serge Demeyer. 2018. "Dynamic Mutant Subsumption Analysis using LittleDarwin". *CoRR* abs/1809.02435. arXiv: 1809.02435. <http://arxiv.org/abs/1809.02435>.

- Petrovic, Goran, and Marko Ivankovic. 2018. “State of Mutation Testing at Google”. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 163–171.
- Pierce, Benjamin C. 2002. *Types and Programming Languages*. 1st edition. The MIT Press. ISBN: 0262162091.
- Polo, Macario, Mario Piattini, and Ignacio García-Rodríguez. June 2009. “Decreasing the Cost of Mutation Testing with Second-Order Mutants”. *Software: Testing, Verification and Reliability* (GBR) 19, number 2 (): 111–131. ISSN: 0960-0833.
- Purushothaman, Ranjith, and Dewayne Perry. July 2005. “Toward understanding the rhetoric of small source code changes”. *Software Engineering, IEEE Transactions on* 31 (): 511–526. <https://doi.org/10.1109/TSE.2005.74>.
- Rafalko, Maks. 2023. “Introduction - Infection PHP”. Visited on May 10, 2023. <https://infection.github.io/guide/index.html>.
- Sahinoglu, M., and Eugene Spafford. July 1999. “Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing” ().
- Schirp DSO LTD. 2023. “Mutant Source Code”. Visited on May 15, 2023. <https://github.com/mbj/mutant>.
- Stein, Eike, Steffen Herbold, Fabian Trautsch, and Jens Grabowski. 2021. “A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations”. *CoRR* abs/2104.02517. arXiv: 2104.02517.
- Stryker Team. 2023a. “Stryker - Documentation”. Visited on May 10, 2023. <https://stryker-mutator.io/docs/>.
- . 2023b. “Stryker Mutator - Mutation Score Example”. Visited on May 10, 2023. <https://dashboard.stryker-mutator.io/reports/github.com/stryker-mutator/robobar-example/master#mutant>.
- Synopsys, Inc. 2023. “What is Continuous Development and How Does it Work?” Visited on May 10, 2023. <https://www.synopsys.com/glossary/what-is-continuous-development.html>.

Technopedia. 2020. “Dictionary - What Does Programmer Mean?” Visited on May 9, 2023. <https://www.techopedia.com/definition/4813/programmer>.

The Elixir Team. 2023. “The Elixir programming language”. Visited on May 7, 2023. <https://elixir-lang.org/>.

Van Do, Tien, Hoai An Le Thi, and Ngoc Thanh Nguyen, editors. 2014. “Problems of Mutation Testing and Higher Order Mutation Testing”. In *Advanced Computational Methods for Knowledge Engineering*, 157–172. Cham: Springer International Publishing. ISBN: 978-3-319-06569-4.

Wichmann, B., A.A. Canning, D.L. Clutterbuck, L.A. Winsborrow, N.J. Ward, and William Marsh. April 1995. “Industrial perspective on static analysis”. *Software Engineering Journal* 10 (): 69–75. <https://doi.org/10.1049/sej.1995.0010>.

Williams, Laurie A. 2010. “A (partial) introduction to software engineering practices and methods”, 60–61.

Winters, T., T. Manshreck, and H. Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O’Reilly Media. ISBN: 9781492082767.

Woo, Jia Hao. 2021. “AppSignal - An Introduction to Metaprogramming in Elixir”. Visited on November 27, 2023. <https://blog.appsignal.com/2021/09/07/an-introduction-to-metaprogramming-in-elixir.html>.

Wu, D., M.A. Hennell, D. Hedley, and I.J. Riddell. 1988. “A practical method for software quality control via program mutation”. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, 159–170. <https://doi.org/10.1109/WST.1988.5371>.

Zhang, Jie, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. “Predictive Mutation Testing”. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 342–353. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery. ISBN: 9781450343909. <https://doi.org/10.1145/2931037.2931038>.

Zilberfeld, Gil. March 2014. “You Can’t Be Agile without Automated Unit Testing”. *Better Software* 2014-02 ().