Investigating the Utilizability of Memoization for Mutation Testing

Computing Science Master Thesis

UTRECHT UNIVERSITY

Department of Information and Computing Science

November 5, 2025

First examiner:
Dr. Wishnu S. B. Prasetya
s.w.b.prasetya@uu.nl

Second examiner:
Dr. Marco Vassena
m.vassena@uu.nl

Author:
Jona R. Leeflang
j.r.leeflang1@students.uu.nl

Company Supervisor:
MSc. Rinse van Hees
rinse.vanhees@infosupport.com

Abstract

Software developers often write tests to verify that their programs behave as expected. However, it is difficult to assess the completeness and robustness of their test suite. Mutation testing is a powerful technique that is used to analyse and improve the quality of a test suite. It works by introducing small faults, called mutants, into a program and checking whether the test suite can detect these. The relative number of mutants caught is used to calculate a mutation score that indicates the quality of the test suite. Developers can use mutation analysis to examine the test results and identify which parts of the test suite can be improved.

A major drawback of mutation testing, however, is the substantial time required to perform it. Although many advances have been made on the computational cost reduction of mutation testing, its efficiency remains suboptimal.

This thesis investigates to which extent memoization can be utilized as a computation cost-reduction strategy for mutation testing in languages with stateful values. For this purpose, we have implemented memoization for a mutation testing tool named Stryker.NET. Our approach uses a shared memory space to maintain and exchange memoized state across multiple independent processes.

We performed an empirical experiment on 15 real-world projects. We found that memoization using serialization can be utilized on roughly 4-22% of methods and expressions in a program. Our results indicate a varying degree of effectiveness across different projects when observing the ratio of memoization cache hits.

Acknowledgements

I would like to express my deepest gratitude to my first supervisor, Dr. S.W.B. Prasetya, for his guidance and support throughout this project. Without our regular meetings to brainstorm ideas and discuss research obstacles, I would not have been able to complete this research. I am also greatly thankful to my second supervisor, Dr. M. Vassena, for his openness during the planning of my thesis defence. I could not have undertaken this journey without the frequent supervision and help of my company supervisor, R. van Hees, MSc., from Info Support. Our weekly meetings and your help finding experts in the company have helped me a great deal during this research project.

Special thanks to N. Thissen, MSc., who has helped me as a process supervisor. Our talks and walks helped me clear my thoughts and plan ahead when I had a lot on my mind. I am also thankful to my fellow and research students with whom I could frequently share my research with and who always had valuable feedback to give. Appreciation should also go out to M. de Roos, MSc., an employee who has previously performed research in the same field, whose work helped guide me forward. I would like to extend my sincere thanks to the employees at Info Support, who made the internship a pleasant experience and provided much support to all research interns both professionally and intellectually.

Last but not least, I would like to express my gratitude to my parents and friends for their unwavering emotional and intellectual support throughout this journey. Their proof reading helped me to improve this thesis and make it more approachable to read.

Contents

1 Introduction **3**

2 Background **5**

2.1 Mutation Testing 5

2.2 Memoization 9

3 Related Works **11**

3.1 Cost Optimization Techniques 11

3.2 Memoizing Techniques 13

3.3 Data Sharing for Memoization 15

3.4 Guidance for Experiments using Mutation Testing 17

4 Approach **18**

4.1 Memoization Design 18

4.2 Detecting Memoizability 20

4.3 Sharing Memoized State 21

5 Implementation in Stryker.NET **22**

5.1 Overview of Stryker 22

5.2 Injecting Memoization 24

6 Experimental Setup **27**

6.1 Assessment Criteria 27

6.2 Evaluation Projects 28

6.3 Test Environment 29

7 Results **31**

7.1 Quality of Testing 31

7.2 Utilizability and Efficiency 32

7.3 Performance 35

8 Discussion and Conclusion **37**

8.1 Discussion 37

8.2 Conclusion 38

8.3 Future Work 38

Bibliography **43**

1. Introduction

Software testing is done to validate whether the software satisfies the expectations of developers and users. However, how can we be sure that the tests we use detect all mistakes a programmer might make? We need to validate the test suite on its quality and effectiveness to know if they are well written. A well-known and reliable method to do this is mutation testing. It works by introducing small semantic changes into a programme to mimic faulty code. Running these mutated programmes against the test suite can give insight into bad performing test suites. Analysis of mutations that are not caught by the test suite can reveal weaknesses or gaps in test cases.

A major drawback of mutation testing is the time required to perform it. For an arbitrary programme, up to thousands of mutations can be generated. Each mutation of a programme has to be tested using the test suite. This large number of test runs performed results in a considerable mutation testing runtime. Reducing the time it takes to perform mutation tests has been an active research topic for decades. Improvement in hardware and implementation has made mutation testing from a novel concept into a practical tool for developers. Modern day mutation tests can be completed within a reasonable time. However, they are still far from being considered fast. When dealing with large projects or poorly designed test suites, execution time may increase significantly, with the mutation testing process sometimes taking tens of minutes to complete. This considerable execution time discourages developers from using mutation testing as a tool in their quality control process.

Recent years have seen increasing research interest in how memoization interacts with mutable and stateful programme behaviour. Memoization is a technique that has been used in different computational areas to reduce redundant computation by storing and reusing computed values. Research by Stoffers et al. Stoffers et al. (2016) demonstrated that automated memoization can drastically reduce redundant computations in large-scale simulations, achieving speed-ups up to 75×. More recently, Ghanbari and Marcus Ghanbari and Marcus (2022) proposed a memoization approach for mutation testing in Java, showing promising improvements in execution time. However, their study also highlighted practical challenges, such as analysis overhead and limited effectiveness in certain real-world projects. The work of Stoffers et al. and Ghanbari and Marcus inspired the idea of exploring how well memoization techniques can be adapted to improve the efficiency of mutation testing under realistic conditions.

The general research question guiding this thesis is: *How can memoization be used in mutation testing to reduce duplicate computations of non-mutated code?* To address this question, we investigate several more specific sub-questions:

Implementation *How can memoization be implemented in a programme that is subjected to mutation testing?*

Feasibility *Is it possible to automatically identify code in a (mutated) programme that can be safely memoized?*

State integrity *Is it possible to prevent unintended modifications of stateful memoization values during mutation testing in a programming language with side effects, such as C#?*

Data sharing *How can a memoized state be efficiently stored and shared between multiple separate executions of mutated programs?*

By answering these questions, this thesis aims to determine both the feasibility and effectiveness of utilising memoization in mutation testing. We use Stryker (Info Support, 2025), a widely adopted and actively maintained mutation testing framework that supports .NET, JavaScript, and JVM languages, as our reference mutation testing tool.

We begin this thesis by providing background information to establish a general understanding of the concepts of mutation testing and memoization. Next, we set out a series of related works that discuss research related to existing computational cost reduction techniques for mutation testing, as well as present methods of memoization and sharing state across processes. These works establish the foundation for our memoization approach for mutation testing. We also describe practical details of Stryker.NET, the mutation testing tool we used as the baseline for our approach. Following this, we empirically evaluate our approach on 15 real-world projects to quantify its utilization and impact on mutation testing performance. The thesis concludes with an analysis of these findings and recommendations for future research.

2. Background

2.1 Mutation Testing

Mutation testing is an approach to measure test effectiveness. It was first introduced by DeMillo et al. (1978). Mutation works by generating altered versions of a programme called mutants that contain artificially injected faults. The idea behind mutation testing is that the small artificial faults represent mistakes a programmer can accidentally make (Jia & Harman, 2011). In the field of mutation testing we often assume the Competent Programmer Hypothesis (CPH) (DeMillo et al., 1978), which states that most programmers are competent enough to write almost correct code. Therefore, it is very likely that any faults in the actual code are small and local. Figure 2.1 shows a simple example of mutated code.

```
1 // Original Code
2 var x = 5 * 2
3
4 // Mutated Code
5 var x = 5 / 2
```

Figure 2.1: Multiplication expression mutated to division expression using Arithmetic Operator Replacement (AOR).

The testing is done by running all mutants against the test suite. If a mutant does not pass the tests it is considered caught or *killed*. If it passes all the assertions, then it was not caught and is considered to have *survived* the test suite. Initially, mutation testing was just a theoretical concept to evaluate test suites. Neither hardware nor mutation testing algorithms were advanced enough to perform the mutation tests in an acceptable time. The number of injected fault a programme can contain is enormous. Therefore, it was too slow and impractical to use in actual development.

The first publicly available mutation testing framework was released in the late 1980s by Choi, DeMillo, et al. (1989). Mothra, a C-based mutation testing framework intended for academic use, marked the start of practical implementations. In subsequent years, better and faster frameworks were released, such as MuJava (Ma et al., 2005), PITest (Coles, 2015), and Stryker (Info Support, 2025). Newer releases pushed better performance. However, mutation testing is still more time consuming than many developers and organisations are willing to invest in their projects.

Compared to other test quality metrics, suhc as line and branch coverage, mutation testing provides developers with more detailed information. Standard coverage metrics only identify what code is less covered by the tests. Mutation testing promises to provide the developer with enough data to be able to analyse a programme to find real faults. By analysing the results of a mutation test, a developer can effectively spot where code is not sufficiently tested as well as identify what type of code faults are not detected by the test suite.

2.1.1 Mutation Generation Policies

A mutation can be any change in the code. Commonly used types are expression and statement modifications. These types can be classified under different mutation generation policies, see Table 2.1. The

choice of policies used has a large influence on the number of mutants and test results.

Many mutation testing policies are language-agnostic, as they leverage basic operations regular to nearly all programming languages. Commonly used policies are Arithmetic Operator Replacement (AOR), Absolute Value Replacement (AVR), Relational Operator Replacement (ROR), Logical Operator Replacement (LOR), and Statement Replacement (SR).

Object-oriented policies also exist (Ahmed et al., 2010; Ma et al., 2002; J. Offutt et al., 2006), such as Explicit Parent Constructor call deletion (IPC) and Access Modifier Change (AMC). These policies are often related to access modifiers, inheritance, and types. Not all object-oriented policies can be used in strongly typed languages like C#.

Some policies are programming language or library specific. For example, JavaScipt has both an equality (==) and strict equality (===) operator. Languages like C#, C++, and Python do not share this strict equality operator. Vice versa, a mutation policy for .NET's LINQ methods (e.g. `.First()/* Mutates to */.Last()`) cannot logically be used in JavaScript.

The effectiveness of each policy differs. For some scenarios, a specific policy may serve little purpose. If so, they can be omitted from the set of policies used (Fernandes et al., 2017). Reducing the number of applied policies is important, as the number of mutants can rapidly grow with the number of policies.

Table 2.1: List of all Stryker.NET mutation types, their names and example conversions of code to their mutated counterparts.

<table>
<tr><th>Mutation Type</th><th>Mutator Name</th><th>Example Conversion</th></tr>
<tr><td>Arithmetic Operator</td><td>arithmetic</td><td>a + b ↔ a - b, a * b ↔ a / b</td></tr>
<tr><td>Equality Operator</td><td>equality</td><td>a == b ↔ a != b, a > b ↔ a < b</td></tr>
<tr><td>Logical Operator</td><td>logical</td><td>a && b ↔ a || b</td></tr>
<tr><td>Boolean Literal</td><td>boolean</td><td>true ↔ false, if (cond) ↔ if (!cond)</td></tr>
<tr><td>Assignment Operator</td><td>assignment</td><td>x += y ↔ x -= y, x *= y ↔ x /= y</td></tr>
<tr><td>Update Operator</td><td>unary</td><td>++i ↔ -i, !a ↔ a</td></tr>
<tr><td>Bitwise Operator</td><td>bitwise</td><td>a & b ↔ a | b</td></tr>
<tr><td>Unary Operator</td><td>unary</td><td>~i ↔ i, -i ↔ +i</td></tr>
<tr><td>Null-coalescing Operator</td><td>nullcoalescing</td><td>a ?? b → b ?? a</td></tr>
<tr><td>Checked Statement</td><td>checked</td><td>checked(2+4) → 2+4</td></tr>
<tr><td>Initializer</td><td>initializer</td><td>new List<int> {1, 2} ↔ new List<int> {}</td></tr>
<tr><td>Collection Expression</td><td>collection</td><td>[1, 2, 3] → []</td></tr>
<tr><td>Block Removal</td><td>block</td><td>{ doSomething(); } → {}</td></tr>
<tr><td>Statement Removal</td><td>statement</td><td>doSomething(); → (removed)</td></tr>
<tr><td>String Literal</td><td>string</td><td>"hello" → ""</td></tr>
<tr><td>String Method</td><td>string-method</td><td>str.Replace("a","b") → str</td></tr>
<tr><td>LINQ Method</td><td>linq</td><td>All(...) ↔ Any(...)</td></tr>
<tr><td>Math Method</td><td>math</td><td>Math.Sin(x) → Math.Cos(x)</td></tr>
<tr><td>Regex</td><td>regex</td><td>"[A-Z]+" → "[A-Z]+?</td></tr>
</table>

2.1.2 Mutation Scores

Performing a mutation test with a set of policies generally results in a mutation score. The score is based on the number of mutants killed and survived. The higher the ratio, the more mutants are killed, and the better the test suite is considered to be.

The simplest calculation for a mutation score is the Raw Mutation Score (RMS). This calculation is straightforward and only deals with the number of mutants killed and survived. Any other context is ignored during this calculation, including mutant equivalence.

$$RMS = \frac{\text{killed mutants}}{\text{total number of mutants}}$$

2.1.2.1 Mutant Equivalence and Redundancy

Mutants in the set of mutants generated for mutation testing can be semantically equivalent to each other (Budd & Angluin, 1982). They exist in two types. The first type, equivalent mutants, are mutants that are semantically equivalent to the *base programme*. All mutants that behave the same as the base programme can be considered a valid variant of it. A real-world example of this would be two developers who implement the same algorithm in a different way. They are syntactically different, but semantically result in the exact same output. The second type, redundant mutants, are mutations that are semantically equivalent to *each other*. Similarly to how equivalent mutants can be valid variants of the original, redundant mutants can be valid variants of each other. See Figure 2.2 for exmaples of the two semantically equivalent types of mutants

```
1   // Original Code        // Equivalent Code
2   var x = 2 * 2           var x = 2 ^ 2
3
4   // Mutated Code         // Redundant Mutated Code
5   var x = 2 / 2           var x = 2 >> 1
```

Figure 2.2: Examples of equivalent mutants (equivalent to the original program), and redundant mutants (equivalent to another mutant).

Both of these can be considered semantically equivalent mutants with respect to the mutation score. Removing a semantically equivalent mutant from the equation does change the mutation score. However it does not meaningfully impact the information that can be taken from the actual tested mutants.

In a theoretical setting, the context of equivalence is important to our calculation of mutation score. Taking into account the equivalent mutants, the number of mutants equivalent to the base programme can be subtracted from the total number of mutants (A. J. Offutt & Untch, 2001). Because of the semantic equivalence, they are valid programmes that cannot fail the test suite. Therefore, they should not be killed and will not contribute the number of killed mutants. Based on the equivalent mutants, a Prorated Mutation Score (PMS) can be calculated.

$$PMS = \frac{\text{killed mutants}}{\text{total number of mutants} - \text{equivalent mutants}}$$

2.1.2.2 Equivalence Classes

The (redundant) mutants that are equivalent to each other fall into the same equivalence class. A class is a set in which all mutants are considered equivalent versions of each other. Mutants in this set will give the same results for any of the test inputs. Therefore, only one mutant of each set is needed to verify if any of them survive or not. Testing all mutants of an equivalence class contributes to redundant computational cost during mutation testing. The problem is that they cannot be easily identified. Budd and Angluin (1982) show that the problem of mutant equivalence is undecidable in the general case. However, research shows that there are many different approximation and estimation methods.

Another alternative calculation is the equivalence-based mutation score (EMS) (Marsit et al., 2021). With this calculation, a very concise score can be made. This is because the number of mutants is already reduced to its minimal number of equivalent classes.

$$EMS = \frac{\text{equivalent classes killed}}{\text{total number of equivalent classes}}$$

2.1.2.3 Mutation Score in Practical Context

Mutation testing tools function in a practical setting, not just a theoretical one. Practical context presents more logic and limitation to the mutation testing process. Mutating loops or recursive calls may introduce timeouts. Some mutations may be ignored and not run if specified by a developer or filtered out by logic such as removing equivalent mutants. Errors can also occur when compiling or running mutants. Table 2.2 shows a common set of completion statuses that a mutant might have after mutation testing.

Calculating a mutation score based on these statuses requires us to know which ones are relevant to the score and which are not. Errors and ignored mutants can be considered irrelevant. Errors occur due to invalid code. Therefore, we can consider these mutations of the programme to be not in the set of valid programmes. Ignored mutants have either been manually decided to be irrelevant by a developer or programme logic, thus do not to be accounted for in the mutation score. All other statuses indicate relevance to the tests detecting invalid mutations and, therefore, to the mutation score.

Table 2.2: Completion statuses of mutants in Stryker.NET, including a description and whether they are included in the mutation score calculation.

Status	Description	Included in Mutation Score
Killed	At least one test fails on mutant	**Yes**
Survived	Mutant passes all tests	**Yes**
No coverage	No test coverage, trivially survives	**Yes**
Timeout	Mutant exceeded the calculated and configured timeout	**Yes**
Ignored	Mutant ignored, due to configuration or mutant filtering	No
Compile error	Mutant caused compile error	No
Runtime error	Mutant caused runtime error	No

2.1.2.4 Insights of a High Mutation Score

In mutation testing, the mutation score quantifies the effectiveness of a test suite by measuring the proportion of mutants that survived compared to the total generated. Although a higher mutation score indicates a more solid test suite, achieving a 100% mutation score generally does not occur (Jia & Harman, 2011). Jia et al. claimed that, based on empirical results, projects contain between 10-40% equivalent mutants. In addition, the 80/20 rule (also known as the Pareto Principle) (Walkinshaw & Minku, 2018) in regard to software testing is often used to reason about the amount of justifiable effort to put into testing. Approximately 80% of faults can be found using 20% of the effort. Based on the fact that mutation testing is expensive to perform and equivalent mutants likely exists for a program, the relative effort top aim for 100% is not realistic. Therefore, it is acceptable to aim for a mutation score around 75-80%.

The main reasons for this are the equivalent mutants and other contextual complexities involved in mutating code. As equivalent mutants are generally not identified, a lower score is obtained. Similarly, practical complexities such as errors and timeouts affect the way scores have to be calculated.

What is more important than reaching full coverage is understanding why mutants survive. Surviving mutants often reveal weaknesses in the test suite and the design of the code itself. Therefore, the mutation score should not be interpreted as an absolute goal, but as a qualitative indicator of the test suite.

2.1.3 Mutation Analysis

The most valuable aspect of mutation testing is the ability to analyse its results. A mutation score is merely an indicator of the quality of the test suite. Analysis can show which code might suffer from poor or flaky testing. Insight into the locality of survived mutants is an important aid for a developer to improve the code and its tests. Reports or interactive dashboards are used to present these details in a consistent and clear way. They show information on how many mutants survive and what type of mutations the test suite does not detect. Without a mutation report, mutation testing is not very useful. It should be used as a tool for the programmer to help design a better and more complete test suite.

An example of a generated interactive report by Stryker is shown in Figure 2.3. In the figure, the number of mutants that resulted in each of the statuses mentioned in Table 2.2 is shown per file and subdirectory.

862 57 66

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	87.51	93.80	860	57	2	66	280	0	71	862	123	1336
Books	88.72	97.52	118	3	0	12	53	0	13	118	15	199
Classes	78.44	88.47	353	46	0	51	126	0	38	353	97	614
Common/ DojoException.cs	N/A	N/A	0	0	0	0	0	0	0	0	0	0
Extend/ Collection.cs	100.00	100.00	3	0	0	0	2	0	3	3	0	8

Figure 2.3: Mutation report (interactive dashboard) generated by Stryker.NET. The columns display the number of mutants in each result status for every file or subdirectory (see Table 2.2 for an overview of the statuses).

Reports can provide additional contextual information, such as information about the type of mutation and covering tests as shown in Figure 2.4. This context can include information on the number of tests covering a mutant. It can also include the type of mutations that survived the test suite. This information helps a developer to know where to look in his codebase to analyse and improve it.

2.2 Memoization

Memoization is a technique first mentioned by Michie (1968). It is a technique to reduce the number of redundant computations. It works by storing computed values and reusing those values instead of recomputing them. It is particularly used in recursive functions and algorithms.

Memoization is used to improve many mathematical procedures that rely on recursive logic and re-

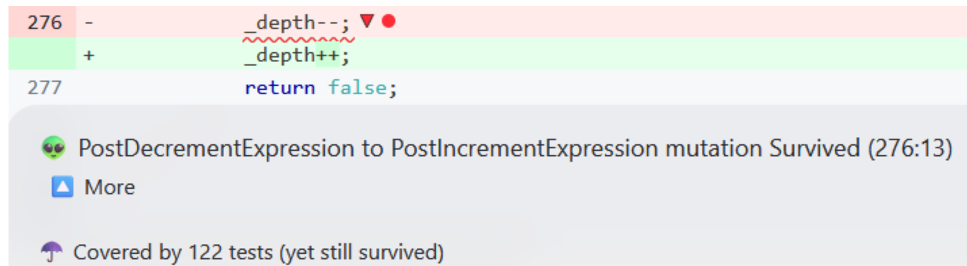

Figure 2.4: Stryker.NET view of mutated code and whether it survived the mutation test. Additional context is shown with mutant type and test coverage. Red means mutant survived.

peated execution. The Fibonachi sequence ($F(n) = F(n-1) + F(n-2)$ with $F(0) = 0$ and $F(1) = 1$) is a prime example of a mathematical formula that suffers from $0(n^2)$ recursive calls (Boyer & Hunt, 2006). Memoization transforms it into a linear $O(n)$, as it stores the results of each previously computed n.

Memoization has since been adopted in many different computational fields, such as software algorithms (Sun et al., 2023), computational biology (Fornace et al., 2020), and computer simulation (Stoffers et al., 2018). The use cases of memoization in these field show a common pattern of expensive and repeated computations, as well as the need to get around memory limitations of large data sets and intermediate states.

3. Related Works

3.1 Cost Optimization Techniques

Mutation testing performance has been actively researched for multiple decades. Several methods for solving different performance problems of mutation testing have been proposed. Many of these approaches can be combined and utilised simultaneously. Thanks to that, the duration of mutation testing has decreased from days to hours to minutes. Ideally, research on improving the performance of mutation testing will continue until any arbitrary programme can be tested in seconds. We will now summarise some of the most important and widely used methods. We consider their contribution and discuss some critique we have on their work.

Different categorisations for cost reduction techniques have been proposed. The most widely referred to in research are *do fewer*, *do smarter*, and *do faster* (A. J. Offutt & Untch, 2001). In a more recent literature study, Pizzoleto et al. (2019) argue that these labels are not specific enough for modern cost reduction techniques, as they often fall into more than one category. They proposed a new set of six categories for cost reduction strategies based on an analysis of the existing literature. The categories are more distinct, reducing overlapping classifications. The categories are: *'Reducing the number of mutants'*, *'Automatically detecting equivalent mutants'*, *'Executing faster'*, *'Reducing the number of test cases or the number of executions'*, *'Avoiding the creation of certain mutants'*, an *'Automatically generating test cases'*.

Many surveys and literature reviews have been done on mutation testing cost reduction (Jia & Harman, 2011; A. J. Offutt & Untch, 2001; Papadakis et al., 2019; Pizzoleto et al., 2019; Usaola & Mateo, 2010). Most works use the 3-category classification when discussing cost reduction techniques. By far the most widely referred survey is the work of Jia and Harman (2011), who have published an extensive survey report reviewing 217 research papers related to mutation testing cost reduction techniques. Another survey on mutation testing advances by Papadakis et al. (2019) presents many techniques, as well as an analysis of mutation-based test evaluation. Moreover, they reveal a growth in the adoption of mutation testing as an assessment method for empirical research, highlighting the need for reliable and efficient mutation testing tools to support large-scale studies. More focused literature studies have also been done on subdomains of mutation testing, such as equivalent mutants (Madeyski et al., 2014), and higher order mutation (Silva et al., 2017). These subdomains tackle the theoretical landscape of software mutation.

Building on these survey studies, the following sections focus on specific techniques and methods proposed to reduce the cost of mutation testing. Each subsection presents approaches addressing different aspects of the problem, ranging from test suite reduction to optimisation strategies, providing a detailed view of practical solutions that have been explored in works from the last five decades.

3.1.1 Test Suite Reduction (TSR)

A simple yet effective idea to reduce computational cost is to minimize the number of tests executed. Normally, during a mutation test run, each mutant is run against the test suite. However, by only selectively running test cases per mutant, we can reduce the time it takes to test it. There is a positive

correlation between the size of a test suite and the speed-up gained from running only a subset of the test suite. Different approaches exist for test suite reduction, but not all of them can be considered adequate (Coviello et al., 2020). Most research on test suite reduction focusses on optimising the test suite. However, for mutation testing, TSR is more effective when used in a dynamic manner. Each mutation is different and may only require a subset of the tests to be performed. Coverage-based methods for the selection of the test subset exist (Fraser et al., 2008), which take into account the coverage of each test. Mutation testing tools often validate a test suite by running it once before mutation to ensure that the original programme passes. Coverage metrics can be collected to find test suite subsets per mutation. Only covering tests are needed to evaluate a mutant. The other non-covering tests should trivially pass, as mutants should have no impact on them. There are also greedy algorithms for finding such subsets based on coverage, as described in the works of Jehan and Wotawa (2023).

3.1.2 Higher-Order Mutation (HOM)

Commonly, mutants consist are a First-Order Mutation (FOM), which is a single mutated unit such as an operator. A relatively new method of mutating programmes is the use of Higher-Order Mutations (HOM). First mentioned in a seminal paper (Jia & Harman, 2008) and later proposed as a feasible approach (Jia & Harman, 2009), this method focusses on combining multiple FOMs to create more complex and difficult-to-kill mutants. A subset of HOMs are strongly subsuming (SSHOM) which can reduce the test effort and increase the test effectiveness (Jia & Harman, 2009). HOMs mostly serve as useful way to analyse programmes and as the number of HOMs grows exponentially with its order. Often greedy algorithms based on (meta)heuristic search (Wong et al., 2020) or optimisation algorithms such as genetic evolution (Abuljadayel & Fadi Wedyan, 2018; Langdon et al., 2010) are used to find the optimal subset of HOMs.

In an earlier section, we mention CPH (DeMillo et al., 1978), a common assumption in mutation testing research that says that most programmers are competent enough to write almost correct code. Just et al. (2014) argue, based on a statistical experiment, that certain HOMs simulate more realistic and subtle developer mistakes consisting of combinations of logic and value errors. SSHOMs are the type of HOMs that cover such subtle cases because the FOMs it consists of overlap on their covering test cases.

3.1.3 Mutant Schemata and Simultaneous Mutation Testing

Another method of cost reduction is mutant schemata (Untch et al., 1993). Mutant schemata is a form of programme encoding. Compiling a programme can take between seconds and minutes to complete, depending on its size. This approach optimises the overhead of compiling a programme per mutant by encoding all mutants into one programme. Using a flag or environment variable, a mutant can be activated and tested. The same programme can be reused and sometimes kept in memory, significantly reducing the execution time for compiled languages. Interpreted languages can also benefit from the reduced memory usage of a single mutated programme.

Mutant schemata is further improved by de Roos et al. (2024). They combined mutant schematics with selective testing to reduce test runs. By analysing test coverage, they can identify mutant groups that can be executed independently of each other. These mutants do not overlap on any test cases or programme logic, allowing them to be tested at the same time. The main advantage of this approach is that less total test runs are needed, reducing the number of test runners that need to be initialised.

3.1.4 Parallelization

An established and effective optimisation technique is parallelisation. Modern hardware supports various different methods of simultaneous execution of a programme. Using parallel execution for mutation testing was first proposed in 1989 (Choi, Mathur, & Pattison, 1989), by using a parallel machine to execute a mutation analysis programme.

Parallel execution of programmes can be done in multiple ways. Modern hardware allows multiple threads of computation to be executed on multiple cores of a processor. Like many programmes, mutation testing frameworks already leverage threads by performing the mutation tests on separate threads. They can do this because mutants are already independently tested.

A more complex variant of this is fork parallelisation. With this approach, an new execution thread is *forked* from the current thread. The new thread continues its computations from the point of forking and uses the same state as that of the original thread. Gopinath et al. (2016) proposed the use of this method as mutation testing cost-reduction technique Instead of parallelising the mutants, their approach parallelises the test execution. Unix and Linux systems currently allow forking operation with Copy-on-Write behaviour, reducing memory copy instructions to only occur when memory is written.

Distributing mutation test runs across multiple hosts is a modern approach to parallel execution (Cañizares et al., 2024; Vercammen et al., 2018). An advantage of this approach compared to traditional single-host mutation testing is that it is very scalable. Using a single host to run all tests is limited by its hardware. A distributed system of multiple hosts hardly suffers from such limitations thanks to sharing the load of computation. This method is useful for larger projects and organisations with a large budget and servers. However, it is less accessible for smaller projects and developers who do not have access to such hardware. A difficult problem in this approach is to find an optimal distribution of mutants so that there is as little as possible time between the first and last host to complete its mutation tests.

3.2 Memoizing Techniques

As mentioned in Section 2.2, memoization is a practical approach to store and reuse computed values. After its mathematical introduction, it quickly got adopted in the world of functional programming (Field & Harrison, 1988).

Its later adaptation in different fields shows practical use for a variety of use cases. The primary motivation for memoization is to reduce the computation time of a programme. Memoization works by storing (intermediate) results to reduce redundant computations. The most common use case for memoization is in recursive logic that often gets the same input arguments. Memoization requires a trade-off between computation time and memory space, as every computation requires memory to store its result. On larger memoization functions, this may cause memory problems.

3.2.1 Selective and Adaptive Memoization

One common use case of memoization is optimising parsers (Norvig, 1991). Parsers can suffer from long recursive loops. Memoization has been shown to vastly improve the performance of parsers. Their repetitive nature causes common repetition of inputs. Memoization has also been shown to help prevent Denial of Service attacks caused by super-linear worst-case behaviour of regular expressions (Davis et al., 2021). Davis et al. (2021) make use of selective memoization (Acar et al., 2003) to optimise the expression matching algorithm without causing too large a memory space cost. Selective memoization

differs from traditional memoization by allowing the programmer or system to explicitly choose which parts of a computation or which function arguments should be cached. This *selectivity* controls how equality between inputs is determined and which intermediate results are worth storing, balancing between computational reuse and memory cost. Later work by Fujinami and Hasuo (2024) extended the approach of Davis et al. by reducing and managing the memo tables more efficiently to further reduce memory consumption. These works show the importance of balancing time and space constraints when using memoization. This task is a complex trade-off, as different programmes can require a different balance between the two.

More advanced approaches to selective memoziation exist. In later work by Acar et al. (2004), an adaptive memoization technique is proposed. Selective memoization decides what is cached, where adaptive memoization determines how cached results can be reused even when inputs are slightly different. Instead of requiring exact matches between a current and previous input, adaptive memoization combines memoization with incremental computation: previously stored results can be retrieved and then efficiently updated (*adapted*) to reflect small input differences. This means that the system can reuse prior computations more flexibly, reducing redundant mappings in the memoization table. They argue that there are many scenarios where slightly different values may result in the outcome.

3.2.2 State Aware Memoization

Most works on memoization are oriented toward functional and effect-free languages (Rito & Cachopo, 2010). Applying memoization to languages with side effects comes with new limitations. These languages are often dependent on an internal state, which makes memoization tricky. Rito and Cachopo propose a Software Transactional Memory (STM) method that applies memoization on transactional states during software execution. Their method is aware of relevant state on top of the input variables. It combines a set of variables read inside the function to determine on which state it depends. And a set of variables written to detect and regulate side-effect operations. This approach is particularly useful in scenarios such as state-space exploration, where computations may step forward and backward through multiple memory frames.

Similar transaction-like memoization has also been adapted in predicate and value analysis methods (Wonisch, 2012; Wonisch & Wehrheim, 2012). Their approach, named Block-Abstraction Memoization (BAM), makes use of memoization to cache previously verified values of a block of statements. Specifically, in their approach they choose to rely only on variables that are locally relevant to a block of statements. Similarly to the write-set of Rito and Cachopo's transactional approach, we can use data flow analysis to carefully detect the state relevant to the input of memoization. Beyer and Friedberger later extended BAM to interprocedural programme analysis, and allow memoization of recursive procedures (Beyer & Friedberger, 2020; Friedberger, 2015, 2021). They use a fixed-point algorithm to unroll and verify programme safety.

Both STM and BAM describe approaches that are desirable for general memoization of programmes. However, they do display limitations on memory usage. These works are used in the context of predicates and logic, which are relatively simple and concise to store in memory. Keeping track of multiple transactional states or memory frames of state can cause a fast growth in memory usage, especially when they would be applied to more unbound systems that use complex and big stateful objects. Their state-aware behaviour, however, is important in finding all relevant inputs to a memoization function.

3.2.3 (Semi-)Automated Memoization

Automating the detection and injection of memoizable code is still a challenge. Stoffers et al. (2018) identify two steps that must be taken to enable automatic and effective memoization. One is the need to identify what code blocks' effort can be saved by memoization. The second is the requirement to alter the code in such a way that the results can be retrieved from the memoization table instead of being recomputed.

The first step of Stoffers et al. presented requires analysis to determine not just the effort to be saved but also which code candidates can be memoized. Research on automated memoization mentions the application of memoization to *memoizable functions*. Typically, its definition is considered to be a pure deterministic function without side effects, where a side effect is any action performed by a function that is visible outside of that function (Finifter et al., 2008). However, in more recent work on compile time memoization, specific cases of impure functions have been considered to be memoizable (Suresh et al., 2017). Suresh et al. mention both read-only pointers and constant values as stateful variables that can be handled like any normal argument. This is a fair assumption, as in most cases a programme does not perform unsafe memory updates to alter such variables.

Nistor et al. (2013) claim 90% of performance bugs involve loops and more than 50% involve nested loops. They mention recursive functions can cause performance bottlenecks in a similar manner. Moreover, Suresh et al. (2017) mention the overhead challenges of hashing, storing, and retrieving results from a memoization table. The smaller a result and its computation is, the more difficult it is to gain performance using memoization. Finding this balance is crucial to identify how much effort memoization can save. Static analysis methods exist to detect expensive methods or threads (Shirazi et al., 1990). The work of Ausiello et al. (2012) describes the use of a k-calling context forest which can identify clues to programme hot spots. More often these methods use a call graph or programme structure graph representation because they provide precise information on dependence and data flow (Jin et al., 2020). Possible hotspots can be collected using these graphs and performance heuristics. However, they cannot always know for sure what code will actually cause the biggest performance bottleneck. Runtime analysis would be required to know that.

The second step of Stoffer et al. regards the manipulation of source code that is required to automatically inject memoization logic. Their approach requires a user-defined annotation to allow semi-automated memoization injection (Stoffers et al., 2016). The work of Besnard et al. (2019) describes the use of a source-to-source compiler to automatically transform code into its memoized counterpart. They use a domain-specific language called LARA (Cardoso et al., 2016) to analyse and automatically detect memoizable functions and inject memoization logic accordingly.

3.3 Data Sharing for Memoization

Implementing memoization within a single process is relatively straightforward, as all storage and retrieval logic operates within the same managed memory space. The main challenge lies in the implementation of a memoization algorithm and managing concurrent access to ensure data consistency. Raposo and Mago Quintao Pereira (2024) propose an interesting approach by implementing memoization using shared pointers. Their idea is to store a computed object in the memoization table and use a shared pointer to allow multiple references to the same object. They make use of a Copy-on-Write (CoW) feature that copies data only when a write operation occurs. Other shared pointers maintain their reference to the unchanged memoized object. This approach can be beneficial, especially if large

objects are stored and often retrieved. However, it also introduces several limitations. Currently, CoW is not supported by all common operating systems. It is only supported for UNIX and Linux systems. Although modern programming languages have compilers for different operating systems, not all code supports execution on them.

Furthermore, this memoization method remains confined to its process' memory boundary, limiting its applicability in multiprocess contexts. This limitation highlights a broader challenge for state sharing. While intraprocess memoization approaches such as CoW efficiently manage shared data, they cannot cross process boundaries. Overcoming this boundary requires mechanisms that enable safe and consistent sharing across independent memory spaces.

3.3.1 Cross-Process Data Sharing

Mutation testing sometimes makes use of separate test runners to perform mutation tests. This ensures that no state is contaminated by alterations from previous runs. Moreover, it allows efficient parallelisation of the mutation test runs. Each test runner can be a separate programme with its own managed memory space. Sharing state across the memory boundaries of multiple programmes is a complex task. One of the reasons is that the memory boundaries are often protected by the operating system. Additionally, each memory space is managed by a different garbage collector or memory deallocation logic. Pointers across these boundaries are generally blocked by the operating system and do not have guarantees of type and memory safety. Memoization across these boundaries requires a more complex and custom approach to maintain consistent data and handle concurrent access to those data.

Mustard and Fedorova (2018) have implemented cross-programme memoization (CPM) for Apache Spark. Their main contribution is an algorithm to compute a unique key based on the user-defined function, input arguments, and communication pattern. This allowed them to implement memoization across Apache Spark processes. A limitation of their work is the lifetime of the data they cache. Knowing the lifetime of data and adapting to it is important to reduce memory overhead. However, with mutation testing, there is no certainty that specific memory will or will not be read at a later point in the testing process. Other methods of sharing state are caching programmes or manual handling of unmanaged memory space (Wanninger et al., 2024). We disregard the use of unsafe code to operate in memory, as this is prone to exploitable memory bugs and security issues (Kedia et al., 2017).

All of these cross-process memory sharing approaches have one thing in common: the need to either copy or serialize the values to comply with the low-level nature of how data is stored in memory. In general, copying or serializing complex programme memory is non-trivial, as memory may include functions, (infinite) data streams, or asynchronous computations that cannot be concretely represented or cloned.

3.3.1.1 Unmanaged Memory

Manually regulating unmanaged memory provides a great deal of freedom in managing one's data. However, it has its drawbacks. Unmanaged memory generally consists of just bytes, with some code allowing read operations for primitive types. Any other data structures require manual implementation of the data transfer and type conversion.

Modern operating systems offer an implementation of virtual memory space named a Memory-Mapped File (MMF) (Microsoft, 2022) or `mmap()` (IEEE, The Open Group, 2018). This is an unmanaged memory space represented as a byte array that offers simple byte and primitive type operations. A key advantage

of MMFs is that they can be easily shared between processes, allowing multiple programmes to access the same data region. Stoffers et al. demonstrate that, by optionally persisting a memoization storage to disk, such shared-memory mechanisms can be extended beyond a single process. This persistence enables the reuse of stored results in multiple simulation runs, extending the applicability of memoization to exploratory parameter studies (Stoffers et al., 2018). However, handling data consistency and concurrent access within MMFs requires manual memory management. This can be addressed through common synchronisation techniques, such as a read-write lock algorithm used in concurrent data structures (Just et al., 2014). These algorithms rely on an atomic lock values value or mutexes to control access to these memory regions. While MMFs enable cross-process sharing, they shift complexity to developers, who must manually ensure synchronisation and data integrity.

3.4 Guidance for Experiments using Mutation Testing

Experiments in mutation testing research can involve numerous implicit assumptions (Papadakis et al., 2019). When these are not explicitly stated, research results may be misinterpreted or inappropriately compared. This is a valid concern, as different mutation testing tools often apply varying mutation operators and policies, which can significantly affect the results.

Papadakis et al. (2019) dedicate an entire chapter to threats to validity in mutation-based experimentation and propose a checklist to improve transparency and reproducibility. Their checklist highlights seven key aspects that should be clearly reported in empirical studies: the selection of mutant operators, the choice of mutation testing tool, handling of redundant mutants, the choice and size of test suites, any reliance on the clean programme assumption, the number of experimental repetitions, and the presentation and granularity of results. These criteria serve as practical guidelines for conducting and reporting mutation testing experiments in a consistent and reproducible manner.

In this study, we adopt the guidelines proposed by Papadakis et al. as a framework to ensure transparency and reproducibility in our experimental design. These guidelines inform how we describe key aspects of our experiment setup, such as the selection of mutation operators and the requirements of the test suite, in Section 6.

4. Approach

Previous chapters have outlined how mutation testing is a useful measure of test quality and how it can be used to analyse and improve a test suite. However, it also suffers from high computational cost due to repeated execution of the same test suite across mutants. In this chapter, we address this problem and present our approach for transforming a project to use memoization.

Repeated test suite executions can cause duplicate executions due to the same inputs repeatedly being passed to a programme. We particularly want to investigate whether these redundant computations can be avoided by utilizing memoization. Memoization allows deterministic computations to reuse previously obtained results instead of recomputing them, potentially reducing the overall test execution time.

The objective of this chapter is to present a conceptual approach for integrating memoization into the mutation testing workflow. We outline the general design of a memoization algorithm, discuss where it could be applied, and describe how the memoized state can be shared safely. We aim to keep this chapter mostly framework-independent and focus on the general concepts that make memoization applicable to mutation testing. The following Section 5 will then describe how these ideas are realised in the Stryker.NET framework.

4.1 Memoization Design

The application of memoization can be done at different levels. For the purpose of this research, we focus on function-level memoization and expression-level memoization. On function-level, we have a body of statements that we can store the resulting return value of. Expressions include computations, such as arithmetic operations and function invocations, that return a value we can store in our memoization storage. Memoizing on expression-level, we define as utilising memoization on assignment expressions to reduce redundant computation on it. The reasoning behind this is that non-assignment expressions do not write state to a variable directly, or do so via side effects. For example, an invocation of a lambda function may not return a value but can indirectly perform logic that alters the programme's state. A memoization algorithm requires three actions to work. Together, they make sure that the storage and retrieval of values is done in a consistent manner. The three actions are elaborated on below.

Store Computed values need to be stored in a memoization table. This table maps unique identifiers to their computed values. A valid identifier is based on its location in the code, its input arguments, and other relevant variables that are read. By converting this information into a concrete identifier, computed values can later be retrieved with it.

Is Memoized Check We need to verify whether a value has already been stored in the memoization table for a given set of input arguments and relevant variables. If it has already been done, we can perform the retrieval action instead of redundantly computing the value.

Retrieve When a value has already been computed and stored, we need to be able to retrieve it. Using the identifier, we need to lookup and retrieve the value from the memoization table.

We combine these three actions into our memoization approach. Our approach is represented in pseudo code below. The CREATEIDENTIFIER operation generates a unique key based on the location in the code, its input arguments and on other relevant variables. This ensures different input states are not wrongfully mapped to the same value.

Pseudo Code 1 Generalized memoization of a computation

```
1:  function GETORCOMPUTE(computation, relevant_input)
2:      id ← CREATEIDENTIFIER(computation, relevant_input)
3:      if ISMEMOIZEDCHECK(id) then
4:          return RETRIEVE(id)
5:      else
6:          result ← COMPUTE(computation)
7:          STORE(id, result)
8:          return result
9:      end if
10: end function
```

Helper operations:
STORE(*id*, *val*) – Saves a computed value in the memoization cache.
ISMEMOIZEDCHECK(*id*) – Checks whether a value is already stored.
RETRIEVE(*id*) – Returns the cached value corresponding to the identifier.
CREATEIDENTIFIER(*comp*, *relevant_input*) – Generates a unique key based on input arguments and relevant variables.

4.1.1 Methods and Expressions

The general memoization logic (Pseudo Code 1) can be applied to both functions and individual expressions within a program.

For functions, this means adding a memoization check and retrieval at the very beginning of the function body. We can replace the body of a function with the memoization logic by encapsulating all statements into a computable block of code such as a lambda function and passing it as an argument to the memoization logic. The first time a computation occurs, the result is stored in the memoization table, and on subsequent calls with the same relevant input, the stored value is returned immediately. Relevant input of a function consists of a combination of its input arguments, and other variables read inside the function body. Pseudo Code 2 illustrates this approach for memoizing a function based on its input arguments and the relevant read variables in its execution context.

Pseudo Code 2 Memoizing a function

```
1: function f_memoized(args)
2:     return GETORCOMPUTE(λ() → f(args), args + read_variables )
3: end function
```

Similarly, expressions such as variable assignments can be transformed to include memoization. Pseudo Code 3 demonstrates how an expression can be wrapped in a memoization check using the same underlying logic. Instead of input arguments, expressions are based on the variables read inside them. These variables from the relevant input are needed to generate unique identifiers for expression memoization.

Pseudo Code 3 Memoizing an expression

```
1: var x ← GETORCOMPUTE(λ() → expr, read_variables)
```

Integrating these transformations from normal code to their memoized counterparts can be done easily. Mutation testing tools traverse a programme's code, via its syntax tree or compiled representation, to inject mutations. Introducing memoization logic into a programme can be seamlessly integrated with these traversal methods. This way, we do not traverse it more times than needed, and we have all the context information about the mutation process to use. In some cases, in particular when the types or dependencies of variables and functions cannot be fully determined from the syntax alone, access to a semantic model (e.g., via a compiler API) may be required to accurately infer all relevant information.

4.2 Detecting Memoizability

Not all computations can be safely and correctly memoized. Hence, prior to applying memoization it is important to determine which computations are suited for memoization. In this section, we outline the criteria and constraints we can use to determine which code segments may be considered a candidate for memoization. A candidate computation can be a function or an expression whose resulting value only depends on a set of inputs and operations that satisfy the constraints below.

Purity and Determinism A computation should produce the same output for the same relevant input variables every time it is computed. We consider the extended definition of purity of Suresh et al. (2017) to include constants and read only memory in the relevant input variables. This includes the requirement that a computation with a certain input should predictably result in the same result. Non-mocked, randomised behaviour is a counterexample of such determinism.

Side-effect free Non-observable side effects should exist in the computation. In context of mutation testing, some side effects may be considered non-observable in respect to the actual programme itself, such as information logging.

State sharing Code may be mutually dependent on a shared state. Memoizing a value and returning it does not automatically update the shared state. Such shared state update has to exist outside of memoization (e.g. assigning a variable), otherwise data consistency is broken.

The goal of these constraints is to create a set of memoizability rules that can be applied during the mutation process to separate candidate computations from incompatible ones. Incompatible computations can either not be memoized in a correct way, or worse, can cause problems in the safety of a programme when executed. These rules help determine, for each candidate, whether memoization is feasible and how to construct the set of relevant input variables. (i.e. function arguments, variables read, and other environmental context) used to generate the unique memoization key.

4.2.1 Automatic Detection of Memoizable Candidates

Automatic detection identifies functions, methods, and expressions that satisfy the constraints described above. Instead of relying on user-defined annotations, our approach uses programme analysis during programme traversal to determine which computations are safe to memoize. Functions are considered candidates if their outputs are fully determined by input arguments and read-only environmental state. Expressions are considered candidates when all the variables they read are known and satisfy the constraints.

Even a computation that appears as safe a candidate may have constraints at runtime that prevent correct memoization. Our approach accounts for these cases by only considering a computation if the types of the output and input variables can be concretely represented. We need to be able to consistently

convert these types to a concrete representation that is either a serialised format or a copyable state. Types that embody concepts such as (endless) streams and computations can generally not be converted into a concrete format.

Some languages require standardised implementations for cloning and serialisation. However, many languages do not have such standards (deprecated) or do not guarantee that serialisation is valid. Because of this, it is not always known at compile time whether a type is serialisable. Runtime verification can be done for these languages to ensure memoization is only utilised on types that support cloning or serialisation.

During mutation testing, the output of a computation may differ depending on which mutants are active. Memoization should not be used when there is any mutant active that could alter the computation's result. By conditionally disabling memoization based on active mutants, we prevent stored values from being corrupted by mutated logic and ensure the correctness of the mutation testing process. Furthermore, we do not desire memoization to mask the possibly faulty execution of mutants that we specifically aim to detect with mutation testing.

By combining static analysis, automatic detection of relevant inputs, and runtime checks, the system can safely identify memoizable candidates without requiring annotations from the developer. The approach balances correctness and automation, ensuring that memoization is applied only where it is safe.

4.3 Sharing Memoized State

The final challenge in our approach is the way to share memoized data. As mentioned in earlier sections, single process memoization can be done relatively easily by sharing a memoization table across threads if needed. The key idea is to maintain a centralised memoization store that is accessible to all relevant computations across threads and processes. This store must satisfy several requirements:

Accessibility All computations that share relevant inputs must be able to retrieve previously memoized values. The memoization store must be accessible to all threads and isolated processes

Consistency The memoization storage must ensure that values are stored and retrieved correctly when concurrently accessed. Updates to the store may not introduce race conditions or inconsistencies.

Using a shared memory space can be a great asset. In this way, the data are separated from the processes, which does not require the processes themselves to host the data. Otherwise, processes would need to handle reallocating the memoized state when they finish and exit their process. However, a challenge is that a shared memory space often does not have information on the structure of the stored data, requiring low-level storage. However, this approach is adaptable to many programming languages.

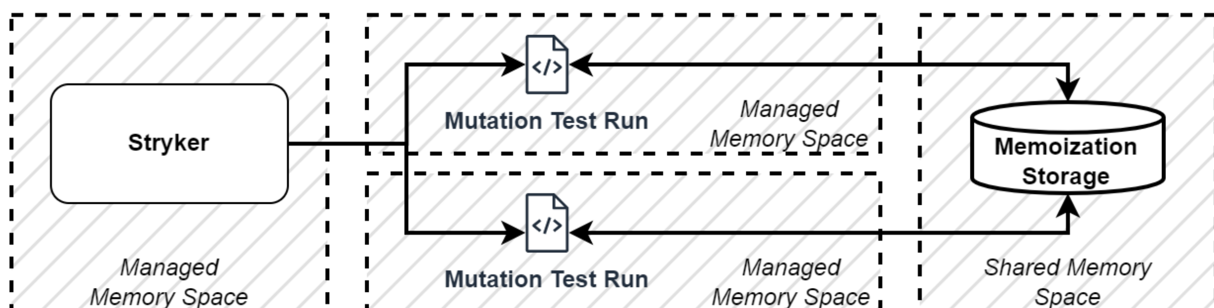

Figure 4.1: Visualization of multiple Stryker test runner instances being disconnected in memory but sharing an unmanaged memory space (where memoization storage resides)

5. Implementation in Stryker.NET

We use the .NET version of Stryker because C# offers strong static typing, advanced reflection capabilities, and fine-grained control over memory and programme execution. These features enable us to implement and integrate memoization at a lower level in the mutation testing workflow, ensuring precise management of state and improved performance. Section 4.3 elaborates on why leveraging these lower-level capabilities is important to our approach.

5.1 Overview of Stryker

Stryker allows different output formats for the mutation report. A terminal progress bar and an HTML report are the standard outputs. Others are also available, such as a detailed JSON file that can be used for analysis.

Internally Stryker follows multiple steps to perform a mutation test. It starts with an initial compilation and test to verify the correct functioning of the original programme. Optionally, this follows with a second run to collect coverage data on the test suite. After this, the project is mutated, and ignored mutants are filtered out. The mutants are tested, after which the desired reports are generated and recorded. See Figure 5.1 for a visualisation of the internal progress of a mutation test.

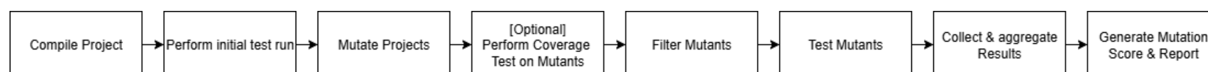

Figure 5.1: Sequential processes Stryker executes when performing a mutation test run.

5.1.1 Internal Architecture

Insight into the inner workings and technical details of Stryker.NET is crucial to our approach and needed to connect memoization to the existing architecture. We present some of the performance improvement methods that have already been implemented in Stryker.NET. We want to be transparent about them, as they can impact the performance differences we measure during our experiment between the original programme and the version with our approach.

5.1.1.1 Syntax Tree Traversal and Mutation

Mutation of a project is done by compiling it into a Roslyn `SyntaxTree`. This tree represents the full syntax of a C# file. Together with a `SemanticModel|` Roslyn builds during compilation, Stryker analyses and mutates the project file by file. A simplified version of the algorithm is presented in Algorithm. 4.

Pseudo Code 4 Simplified Algorithm of Stryker Recursive SyntaxTree Traversal And Mutation

```
 1: Mutators ← set of all mutator classes handling mutation generation
 2: procedure MUTATE(node)
                                   ▷ Top-Down phase: collect mutations for current node
 3:     mutations ← [ ]
 4:     for all mutator ∈ Mutators do
 5:         for all nodeMutation ∈ mutator.MUTATE(node) do
 6:             mutations.ADD(nodeMutation)
 7:         end for
 8:     end for
                                   ▷ Recursive descent: process children (still top-down)
 9:     for all child ∈ node.CHILDREN do
10:         mutatedChild ← MUTATE(child)
11:         node.REPLACE(child, mutatedChild)
12:     end for
                                   ▷ Bottom-Up phase: inject collected mutations
13:     mutatedNode ← INJECTMUTATIONS(node, mutations)
14:     return mutatedNode
15: end procedure
```

First, Stryker works recursively from the top to the bottom of the tree. As it passes each `SyntaxNode`, it generates mutations for that node when any mutator logic exists for it. It does so by going through Stryker's mutators and checking if any mutators match the node's type. It follows this by calling the mutation function on its child nodes, working its way to the bottom of the tree. From the bottom, Stryker works its way back up by replacing the mutated child nodes in the node. This is followed by injecting the generated mutations of the node itself and returning it back to its caller. We end up with a mutated syntax tree that contains all generated mutations.

5.1.1.2 Mutant Activation Mechanism

Stryker makes use of mutant schemata (Untch et al., 1993) by compiling mutants from the mutant generation process into the same mutated programme. All mutants co-exist in the same compiled programme allowing us to use a mutant activation mechanism to dynamically activate mutants. Each mutant is associated with an identifier. During test execution, only the mutants whose identifiers are chosen are activated.

In Stryker.NET, the identifier check is done via an injected `IsActive(id)` call. These checks are injected during mutation, and implemented in slightly different ways depending on the type of mutated syntax. Mutated expressions use ternary logic, statements use if-statements with full bodies. The implementation of `IsActive(id)` is defined in a C# file that is injected into the project files before compiling the mutated programme.

5.1.1.3 Coverage Test & Test Suite Reduction

Optionally, a test coverage run is performed to allow Stryker to implement test suite reduction (Jehan & Wotawa, 2023) per mutant. By tracking the coverage of the tests over the mutants, before performing the actual mutation tests, the test suite can be reduced to a subset of test cases. Mutants without covering

test cases can be disregarded for mutation test runs, as they would trivially survive.

The coverage test run is done by using an alternative implementation of `IsActive(id)` from 5.1.1.2. During the run, all mutants that are reached by the test suite record their mutant identifier. A C# data collector retrieves all identifiers from the test runner. Stryker uses this to generate a mapping from each mutant to their covering test cases. Furthermore, the mutant-testcase mapping is used to identify which sets of mutants are not covered by the same test cases. This allows Stryker to implement simultaneous mutation testing by activating all mutants in a set at the same time during a test run (de Roos et al., 2024). This reduces the number of separate test runner instances needed to perform the test runs. In Figure 5.2, a diagram shows the use of the test coverage information in the mutation analysis workflow.

breaktt

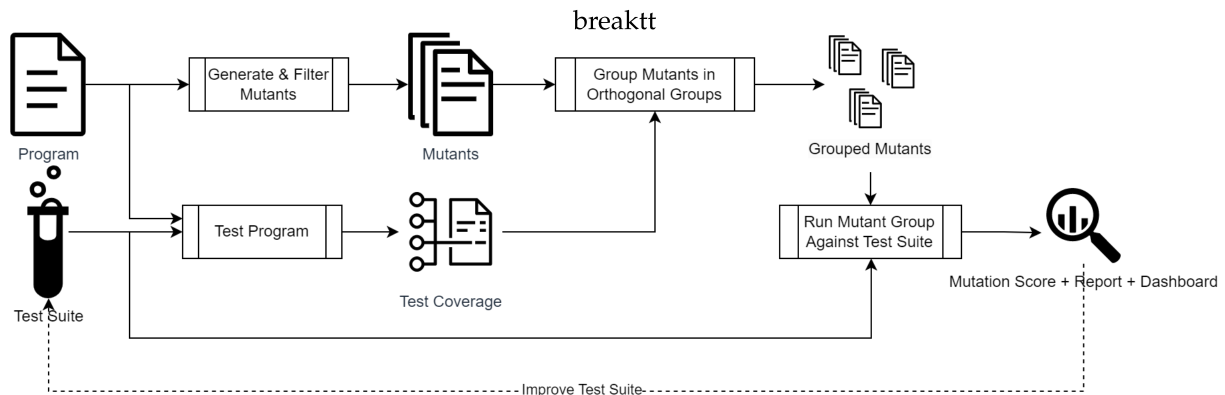

Figure 5.2: High level visualisation of how Stryker.NET uses test coverage information to apply Test Suite Reduction and group mutants that are not covered by the same test.

5.1.1.4 Test Execution Orchestration

When all mutants or mutants groups are created and injected into the syntax trees Stryker starts testing. By default Stryker uses half the threads available to run mutation tests in parallel. This is done to reduce the computation time of testing. After the runs are done, the results are collected, aggregated and reported in the reporters that are configured for Stryker.

5.2 Injecting Memoization

In line with the memoization method proposed in Section 4.1, we implement a new intermediate step during the syntax tree traversal of the mutation process.

During the step on line 13 of Pseudo Code 4, `InjectMutations`, we extend the logic to allow additional code injection. When injecting mutations into the current node, we have the context of the mutated child nodes and the original non-mutated node. Additionally, we can access Roslyn's semantic model, which can be used to infer types and obtain information about the flow of data.

When traversing the syntax tree in Stryker, we can differentiate between the different types of syntax nodes. Stryker uses classes called mutant orchestrators to separate the mutation logic of different syntax nodes from each other. We are interested in the `BaseFunctionOrchestrator` and `LocalDeclarationOrchestrator`. The `BaseFunctionOrchestrator` is the base type for orchestrators that are used for methods and functions. Inside this class, we implement the logic to inject our memoization into the mutated programme. For expression-level memoization, we use `LocalDeclarationOrchestrator` because it orchestrates mutations for statements where variables are assigned.

5.2.1 Generating Unique Identifiers

All memoized values should be uniquely identifiable to prevent values of different memoized computations from being stored under the same key. We create a unique location identifier using the fully qualified name of the method, or the parent method in the case of expressions, together with the code position in the C# file. We use both syntactic and semantic analysis to collect the set of relevant variables that are read inside the computation. Using serialisation, we convert these values into strings and concatenate them with the location identifier to create the memoization identifier. This value is unique to its memoization code and the input variables. We use serialisation, because it is widely used and well supported by most programming languages.

5.2.2 Packing Original Code in Lambda

To model the computation that we described in our approach, we use lambda functions to encase the mutated version of the original programme. This is the code that should be executed if a mutant computation is activated or if no value is stored for the current input arguments. Lambda functions allow easy wrapping of expressions in Roslyn, which is why we chose to use them.

For each memoized computation, we pass its computation lambda, the identifier of the memoization and a predicate that the memoization can use to verify there are no active mutants that can contaminate the memoization store. When Stryker compiles the mutated and memoized programme, the memoization logic is added to the project files, allowing us to invoke the memoization method during testing.

5.2.3 Constraints Limiting Memoization

Before actually injecting the memoization, we perform a series of verification checks based on the syntax and semantics of the computation. We validate the constraints for correct memoization we presented in Section 4.2. We have some additional language specific constraints that computations must satisfy.

5.2.3.1 Compile Time Constraint Validation

In the following list, we outline the types of code that constrain us from using memoization that we can detect during compile time. These can be inferred from the syntax and semantics of the mutated code, without the need for runtime information.

Side-effects A computation may observably alter state outside of the scope of the code that is being memoized. Altering fields or variables that come from an outer scope can cause side effects for logic coming after the memoized code. Using Roslyn's semantic model, we extract data flow information to identify any variables outside the scope of the method or expression being written.

External invocations External sources and APIs cannot be analysed using Roslyn in Stryker because it does not automatically load those assemblies and their semantics into the semantic model.

Void return type Any function or expression that does not return any value (`void`), thus cannot be memoized.

Empty body Functions and methods from getters, setters, and interfaces often do not have implementations and can therefore not be memoized.

Incompatible keywords Some keywords in C# cause our errors when compiling our memoization approach. Specifically, the keywords: `in`, `out`, `yield`, `ref` and `ref struct`.

Async/awaiter Asynchronous code may not always be convertible to a concrete value that can be stored. Moreover, because of external calls, threads, and other logic of async code, there may be side-effects we cannot observe that cause incorrect behaviour when memoized. Awaiter code works in a similar limiting manner.

5.2.3.2 Runtime Serialization

In addition to compile-time detection, we need runtime validation. To use serialisation, our values must support it. There used to be a serialisable attribute used in C#. However, this has long been deprecated, and is no longer a consistent way of ensuring a type is serialisable at compile time. During runtime, we can experimentally collect information about types that are serialisable. The first time some type is computed via a memoization approach, we attempt serialisation. When this fails, we mark that type as not supported and do not attempt further serialisation or memoization on values of that type. If it *is* serialisable, we keep memoizing the computed values of this type.

5.2.4 Sharing Memoized Values

There is limited work on sharing state across memory boundaries in a standardised and connected way. Our approach needs to be accessible and consistent. We use a Memory-Mapped File (MMF) to store our data. This is an unmanaged memory space that can be accessed via the operating system via a string identity. We implemented a custom memoization table that uses an MMF as the backing data structure to store the data. A memory mapped file can be stored in the RAM (volatile) or in a file (persistent). For simplicity's sake, we have chosen to use a persistent file, as this allows more flexibility in storing large amounts of data. Our data structure algorithm uses an operating system mutex to handle concurrent access to the memoization store

Our data structure works like a C# Dictionary with two methods: `Add(id, value)` and `TryGet(id, out value)`. These are the only methods we need for a working memoization implementation. The data structure converts data that are to be stored into serialised bytes, and back to its type when retrieved. Inside the MMF we use a hash map implementation with a custom hash function to theoretically get a near $O(1)$ lookup time. We initialise the MMF before mutation testing and dispose of it afterwards to ensure no memory leaks are created. The implementation of this data structure is injected together with the memoization code from Section 5.2.2.

We deliberately have chosen not to use a large cache tool or library as this imposes versioning restrictions on the programmes that are tested. Our implementation would have to support different API versions. In addition, Stryker would need to identify and inject the version of the library that is supported by the programme under mutation. Another option, unsafe code, imposes project restrictions like libraries by requiring the project settings to have unsafe code enabled.

6. Experimental Setup

In order to perform a validation on the effectiveness and performance of using memoization in mutation testing, a solid setup is required. The evaluation is done through an empirical experiment on 15 real-world projects. Per project, we perform 6 repetitions of mutation tests to improve the accuracy of our results. In our experiment, we make use of two instances of Stryker.NET for mutation testing:

[B] Baseline The original Stryker.NET executable at version 4.5.1, using the .NET 8 runtime. This version is largely kept in its original form, aside from minor additions for data collection on the runtime performance.

[M] Memoized An altered version of Stryker.NET 4.5.1, extended with our memoization approach. This version makes use of our Memory-Mapped File shared memoization store to utilise memoization across mutation test processes. This implementation also collects data on the runtime performance. Additionally, this version collects information on which candidate computations have been memoized, how frequent memoizations are called, and how performant they are.

The goal of this experiment is to collect data on the utilization and effectiveness of memoization. As detailed in Section 3 and Section 4, it is not possible to arbitrarily memoize any piece of code. With the data we collect while running our experiment, we can identify when we can safely memoize and under which constraints we can not.

6.1 Assessment Criteria

In our experiment, we assess three different aspects of our approach in comparison with the baseline. We categorize them as the *quality of testing*, *utilizability and efficiency*, and *performance*.

Quality of testing A factor in cost reduction techniques is making sure that the quality of mutation testing stays good. The project-wide mutation score indicates whether the approach is implemented correctly. In our experiments, we are interested in verifying whether the number of mutants tested, survived, killed, and timed-out remain comparable. We consider a difference of 2% between the baseline and our approach acceptable. In addition, the mutation score should also stay roughly the same to be considered acceptable. There is one exception to this, the timed-out mutants. Because our implementation is not fully optimized, certain mutants may exceed the allowed execution time, leading to timeouts.

Metrics	# mutants created; # killed, # survived, # timed-out

Utilizability and Effectiveness These aspect are the primary focus of this study. We are interested in the number of candidate computations for memoization, but we are also interested in the reasons why the others were not memoized. These give insights on the types of programmes that can be memoized and on the reasons the others can not be memoized. Our implementation uses serialization for storing and copying values to and from the memoization store. Not all data types are serializable. Therefore, we further collect information during runtime to filter out what memoized computations can be serialized and utilized. In addition, we measure how often serializable mem-

oization is actually used. We collect information on the number of calls to the memoization logic. With this we can analyse if there is any memoization code that is not used during runtime. We also want to know how often memoization retrieves a value from the store successfully (hit) and how often it computes not yet stored values (miss). We calculate the total calls to memoization using the hits and misses.

Metrics	# code memoized; # code not memoized (including causes); # hits/misses; # times called

Performance This is an aspect that is important for the speed up this cost-reduction approach could gain. We record the computation time of different steps of the mutation testing process to assess the impact of our approach. Specifically, we measure this for: the initial test, the process of mutation the programme, the mutant coverage test, and the testing of the mutants. We also measure the performance of the memoization logic itself on a smaller scale. These are the time it takes to retrieve and store values, as well as the serialization process.

Metrics	# code memoized; total runtime; mutation time; testing time; memoization (serialize, deserialise, retrieve, store, full process)

6.2 Evaluation Projects

The evaluation and data collection for our research require a diverse set of projects to ensure the robustness of our conclusions. To maintain validity and transparency, we defined three inclusion criteria that each project must meet. Poorly maintained or poorly tested projects could negatively affect the reliability of our results.

- **C1. Open-Source:** Projects should be open-source. The use of publicly available code allows other researchers to verify and replicate our experiment. In contrary, closed-source projects hinder credibility due to limited accessibility.
- **C2. Reliable Test Suite:** Each project's test suite should have sufficient test coverage to indicate adequate validation of correctness. Higher coverage does not directly correlate to quality. However, it does indicate a higher interaction between testing and code. We aim to use projects with approximately 50% or more code coverage. In addition, we disallow projects with flaky test suites to be used. Any test suite that differs more than 1% between repeated runs is too inconsistent for accurate measurements.
- **C3. Category Variety:** The selected projects should represent a diverse range of application domains and computational characteristics. Different types of software tend to produce recognisable mutation patterns. Insufficient diversity in project types may introduce bias into the results. For example, mathematical libraries often contain many arithmetic mutations but few conditional ones, whereas HTTP libraries may exhibit the opposite trend.

We used 15 different projects from a number of different categories of computation types to evaluate the performance of our memoization. These projects have been collected from GitHub and are compatible with Stryker and .NET 8. Aside from category variety they have been chosen based on stars, forks and popularity in the C# community.

Table 6.1: Table of evaluation projects used in the empirical experiment. The columns show information on the number of test cases, mutants generated and size. The projects are ordered by the Lines of Code (LoC) of the projects.

Project	Project Type	Tests Cases #	Mutants Tested #	Size (LoC)	Github Stars #
RepoDB.Core	[Lib] ORM	4 420	4 712	192 299	1.8k
Validot	[Lib] Model Validation	7 348	2 801	50 857	339
CsvHelper	[Lib] CSV Read/Write	1 063	2 733	37 127	5.1k
Enums.NET	[Lib] Enum Utility Methods	224	723	9 398	1.8k
Marsen.DoJo.Net	[Collection] Code Exercises	442	916	8 052	4
MathFlow	[Lib] Math Parsing & Evaluation	166	1 671	7 278	47
Prism.Core	[Lib] MVVM	277	375	4 069	6.6k
SharpPhysics	[Lib] 2D Physics Engine	30	448	3 402	39
SimplifiedSearch	[Lib] Fuzzy String Matching	117	131	3 105	7
ByteSize	[Lib] Byte Representation	106	213	1 514	598
StreamUtilities (Raw.Streaming.Webhook)	[Tool] Live Streaming	71	174	1 435	-
Coravel.Mailer	[Lib] E-mailer	67	99	983	4.2k
NaturalSort.Extension	[Lib] Natural Sorting	35	107	668	201
PolymorphicJson	[Lib] JSON Type Serialization	100	6	246	3
MemoizationBenchmarking	[Lib] Custom Project for Research	114	34	47	-

6.3 Test Environment

Mutation test runs are performed using a Stryker configuration file. In this way, each test run is performed with the same resources. Some projects use an alternative version of the configuration file. Some of the projects require the `.csproj` file in the configuration to work. The path of the configuration file is passed to Stryker as a command-line argument. In the following list, we present the configurations used during the evaluation.

logLevel: info This configuration limits the programme to only output info level logging. Other levels contain more logging that can impact the performance measurement collection.

reporters: [html, json] This sets Stryker to only output HTML and JSON files with mutation score and information at the end of the mutation testing process. (Other reporters can connect to online API's or continually update the terminal, which may impact the performance of Stryker during the mutation testing process itself). We also include our own implemented reporter that writes the runtime measurements to a file.

mutation-level: complete This enables all types of mutants that Stryker can create. Complete is the most rigorous version of mutation levels. Using the complete selection of mutation operators ensures that we have the maximum number of mutants Stryker can create. This can help show how often memoization might be used in relatively expensive mutation test runs. Stryker does *not* do a filter out redundant mutants before running them. It only filters out mutants for specific scenarios and manually disabled mutants.

additional-timeout: 20 000 ms The timeout delay is normally computed based on the normal runtime

of a test run. Beforehand we cannot ensure our approach is not slower that the baseline. As such, we need to build in an additional timeout delay to allow our memoization version to finish its computations.

coverage-analysis: perTest Setting this to *perTest* causes Stryker to use test coverage information to only run the selection of tests that cover active mutants. This is an implementation of TSR in Section 3.

output Used to collect all Stryker output in a centralised location for data analysis.

We use one host to run the tests to keep the results comparable. All test runs are performed separately and without any other programmes running in the background. The host used is a Dell laptop running Windows 11. See Table 6.2 for the specifications of the machine.

Table 6.2: System specifications of device used to run the experiment mutation test-runs

Device	Dell Latitude 5531
Operating System	Windows 11 Enterprise
Architecture	x64
Processor	Intel i7-12800H 1800Mhz 14 cores 20 logical processors
RAM	32 Gb, 4800 MT/s
Disc	Micron 3400 NVM 1024 Gb
.Net Runtime	Dotnet 9.0.305

7. Results

During the experiment, over 150 GB of raw data on performance and memoization usage were collected and subsequently reduced to approximately 25 GB of structured CSV files that contained the information used in our analyses. This processed dataset was analysed and summarized into the tables shown below. In this chapter, we present and interpret the key results derived from these analyses.

7.1 Quality of Testing

The first assessment aspect that we presented was the quality of the testing. For this reason, we collected the data on the mutation scores resulting from both the baseline and memoized versions. Table 7.1 displays the means of the mutation scores of the projects evaluated for both of our Stryker runners (B and M). In addition, it shows the standard deviation of the mutation scores for both runners. The last column (Δ Mutation Score) shows the difference in mutation score between the two runners. These are differences in percentage points, not relative difference (%).

Table 7.1: Mean mutation scores of the evaluated projects for both Stryker versions (B and M), including their standard deviation between repeated runs. Δ *Mutation Score* – Percentage points difference between the Stryker versions.

Project	Mutation Score (B)	Mutation Score (M)	Δ Mutation Score
RepoDB.Core	26.516% (±0.092)	37.273% (±2.084)	10.757 %pt.
Validot	96.942% (±0.000)	98.034% (±0.165)	1.092 %pt.
CsvHelper	64.181% (±0.089)	72.134% (±1.627)	7.953 %pt.
Enums.NET	41.350% (±0.000)	45.105% (±0.545)	3.755 %pt.
Marsen.DoJo.Net	84.975% (±0.000)	85.448% (±0.299)	0.473 %pt.
MathFlow	29.474% (±0.038)	38.612% (±1.349)	9.138 %pt.
Prism.Core	28.454% (±0.000)	28.624% (±0.392)	0.170 %pt.
SharpPhysics	14.398% (±0.000)	14.783% (±0.756)	0.385 %pt.
SimplifiedSearch	94.656% (±0.000)	96.183% (±2.159)	1.527 %pt.
ByteSize	76.151% (±0.000)	76.151% (±0.000)	0.000 %pt.
StreamingUtilities	40.294% (±0.703)	40.996% (±0.485)	0.702 %pt.
Coravel.Mailer	31.150% (±0.226)	31.150% (±0.226)	0.000 %pt.
NaturalSort.Extension	84.545% (±0.000)	89.394% (±5.322)	4.849 %pt.
PolymorphicJson	83.333% (±0.000)	83.333% (±0.000)	0.000 %pt.
MemoizationBenchmarking	30.000% (±0.000)	33.300% (±0.000)	3.300 %pt.

We observe a noticeable variation in Δ Mutation Scores. About two-thirds of the projects have mutation score differences below $< 2\%$, meeting the quality criteria defined in Section 6.1. In contrast, other projects show a large difference in mutation scores, ranging from approximately 4 to 11%. This indicates that the memoized version of Stryker produces incorrect results in some scenarios. About two thirds of the projects show consistent mutation scores, indicating correct handling in most scenarios. On top of that, we find that the standard deviation of the mutation scores in the memoized version of Stryker shows a higher inconsistency in the final mutation score than the baseline. The baseline also demonstrates minor deviations in some results. These deviations may be due to mutants whose execution time is near Stryker's timeout threshold.

Table 7.2 shows detailed information on the mutants tested and the statuses of the mutants after the test run. The second column shows the number of mutants that have been tested by Stryker. Those mutants end up killed, survived, or timed-out after a mutation test run. The remaining columns of the table show the number of mutants that ended up in each of those states. Again, roughly two-thirds of a projects maintain killed mutant difference within 2%. RepoDb is an outlier, as the memoized version kills only two-thirds of the mutants compared to the baseline. In the column of survived mutants we see a similar trend, where approximately 54% of mutants that normally survive, do not. This trend becomes clearer when examining timed-out mutants. There is a massive increase in the number of mutants that timed-out. A bottleneck in our implementation likely causes the increased number of timed-out mutants we observed. In the configurations, an additional timeout of 20 seconds was added to reduce such timeouts, but they seem to not have prevented all of them. In general, the number of mutants that were timed-out instead of killed/survived seems to correlate with the difference of mutation scores shown in Table 7.1.

Table 7.2: Mutants tested, killed, survived, and timed-out per project, ordered by LoC (largest first). Each mutant status column includes the absolute and percentage difference between the original and memoized Stryker runs. *Mutants tested* – Total number of mutants executed during testing. *killed* – Mutants that caused a test to fail (detected). *Survived* – Mutants that passed all tests (undetected). *Timeout* – Mutants that caused the tests to exceed the time limit.

Project	Mutants Tested			Killed			Survived			Timeout		
	B	M	Δ / Δ%	B	M	Δ / Δ%	B	M	Δ / Δ%	B	M	Δ / Δ%
RepoDB	4 724	4 724	(+0 / +0.0%)	2 673	1 793	(-880 / -32.9%)	2 038	949	(-1090 / -53.4%)	12	1 982	(+1970 / +15757.3%)
Validot	2 801	2 788	(-13 / -0.5%)	2 583	2 372	(-211 / -8.2%)	75	45	(-30 / -40.0%)	143	371	(+228 / +159.0%)
CsvHelper	2 733	2 716	(-16 / -0.6%)	2 053	1 874	(-179 / -8.7%)	612	346	(-266 / -43.4%)	68	496	(+428 / +628.1%)
Enums.NET	723	726	(+3 / +0.4%)	514	472	(-43 / -8.3%)	190	145	(-45 / -23.9%)	19	110	(+91 / +487.1%)
Marsen.NetCore.Dojo	916	916	(+0 / +0.0%)	817	823	(+6 / +0.7%)	79	74	(-5 / -5.9%)	20	19	(-1 / -5.8%)
MathFlow	1 671	1 666	(-5 / -0.3%)	1 022	996	(-26 / -2.5%)	606	274	(-332 / -54.8%)	44	396	(+352 / +801.1%)
Prism	305	306	(+1 / +0.3%)	240	241	(+1 / +0.3%)	62	60	(-2 / -3.5%)	2	5	(+2 / +100.0%)
SharpPhysics	448	447	(-1 / -0.2%)	219	216	(-4 / -1.6%)	228	223	(-5 / -2.0%)	1	8	(+7 / +716.7%)
SimplifiedSearch	131	131	(+0 / +0.0%)	124	121	(-3 / -2.4%)	7	5	(-2 / -28.6%)	0	5	(+5 / +0.0%)
ByteSize	213	213	(+0 / +0.0%)	182	182	(+0 / -0.2%)	31	31	(+0 / +0.0%)	0	0	(+0 / +0.0%)
StreamUtilities	174	176	(+2 / +1.1%)	105	107	(+2 / +1.7%)	69	69	(+0 / +0.2%)	0	0	(+0 / +0.0%)
Coravel.Mailer	99	99	(+0 / +0.0%)	71	71	(+0 / +0.0%)	28	28	(+0 / +0.0%)	0	0	(+0 / +0.0%)
NaturalSort.Extension	107	107	(+0 / +0.0%)	90	76	(-14 / -15.4%)	14	9	(-5 / -38.1%)	3	22	(+19 / +638.9%)
PolymorphicJson	6	6	(+0 / +0.0%)	5	5	(+0 / +0.0%)	1	1	(+0 / +0.0%)	0	0	(+0 / +0.0%)
MemoizationBenchmarking	34	34	(+0 / +0.0%)	32	32	(+0 / +0.0%)	2	2	(+0 / +0.0%)	0	0	(+0 / +0.0%)

7.2 Utilizability and Efficiency

The primary results of our experiment relate the question of whether memoization can be utilised, and to what extent. The utilisation data indicate promising potential for memoization. Table 7.3 summarises the total number of computations evaluated for memoization, highlighting the proportion that can be utilised at runtime using serialisation. Next to that are those who were considered not to be memoizable during compile time (*Not Memoizable (%)*). The next two columns show how many were to be memoized (*Memoized (%)*), and how many of those were identified as having serialisable input variables (*Serialized (%)*). This table shows how often in a project there is an opportunity to utilise memoization and theoretically reduce the computational cost of the mutation testing process.

The first thing to note is that the projects have 10 to 56% of their computations memoizable. These percentages are surprisingly high, given the fact that C# is a programming language that contains a lot of state manipulation and side effect behaviour. As discussed in earlier sections, serialisation is an approach that cannot be applied on all data types. If we compare the memoized and serialised columns we can see that for most projects, the serialisation constraint reduces the number of memoiza-

tions we can utilise by up to 80%. However, the serialisable memoization candidates of the majority of the projects get reduced by just 15 to 35%. There are some smaller projects that have memoizations that support runtime serialisation. Excluding these outliers, most projects have 4–22% of computations that are runtime serialisable. `NaturalSort.Extension` is a positive outlier, with approximately 53% of its examined computations supporting serialisation. There appears to be a correlation between project type and the proportion of code eligible for memoization. The type of projects that do a lot with mathematical computations (`MathFlow`, `sharpPhysics`), strings (`SimplifiedSearch`) and simple algorithms (`NaturalSort.Extensions`) utilise relatively more memoization than projects that target HTTP (`Coravel Mailer`, `StreamingUtilities`), model validation (`Validot`), and (data) model manipulation (`Prism`, `RepoDb`). Each memoization call could potentially reduce computation time if executed more efficiently than the original computation. Even 5 to 10% could improve mutation testing runtime noticeably, if they are computationally cheaper than the original computation.

Table 7.3: Number of examined computations being memoized. Projects ordered by descending LoC. *Not Memoized* – Number of computations not memoized. *Memoized* – Number of computations memoized at compile time. *Serializable* – Number of memoized computations that use runtime serializable types.

Project	Total	Not Memoized (%)	Memoized (%)	Serialisable (%)
RepoDB.Core	5 991	3 194 (53.3%)	**2 797 (46.7%)**	470 (7.8%)
Validot	1 361	1 233 (90.6%)	128 (9.4%)	94 (6.9%)
CsvHelper	1 843	1 472 (79.9%)	**371 (20.1%)**	137 (7.4%)
Enums.NET	658	514 (78.1%)	**144 (21.9%)**	**69 (10.5%)**
Marsen.DoJo.Net	563	423 (75.1%)	**140 (24.9%)**	**114 (20.2%)**
MathFlow	1 097	484 (44.1%)	**613 (55.9%)**	**322 (29.4%)**
Prism.Core	665	567 (85.3%)	98 (14.7%)	17 (2.6%)
SharpPhysics	674	401 (59.5%)	**273 (40.5%)**	**94 (13.9%)**
SimplifiedSearch	128	95 (74.2%)	**33 (25.8%)**	**28 (21.9%)**
ByteSize	69	69 (100.0%)	0 (0.0%)	0 (0.0%)
StreamUtilities	361	331 (91.7%)	30 (8.3%)	14 (3.9%)
Coravel.Mailer	67	149 (85.6%)	25 (14.4%)	7 (4.0%)
NaturalSort.Extension	30	14 (46.7%)	**16 (53.3%)**	**16 (53.3%)**
PolymorphicJson	7	7 (100.0%)	0 (0.0%)	0 (0.0%)
MemoizationBenchmarking	30	20.0 (66.6%)	**10.0 (33.3%)**	**6.0 (20.0%)**

Table 7.4 summarises the effectiveness of memoization in the evaluated projects. For each project, the table reports the number of successful value retrievals (hits) and the number of cases where a value had to be recomputed (misses). The final column presents the hit ratio as a percentage.

Overall, the data indicate a relatively high hit ratio for most projects, suggesting that memoization can effectively reduce redundant computations. RepoDb, however, exhibits a comparatively low hit rate, which aligns with the limited effectiveness of memoization observed in Table 7.3. For the remaining projects where memoization was applied, hit ratios are consistently high, highlighting substantial potential for performance improvement through repeated memoization hits.

We also analysed what the reasons were in our evaluation projects to limit memoization during compile time. Table. 7.5 shows the count of each reason we do not memoize during compile time. The counts are split between method-level computations that have not been memoized, and expression-level ones.

The most common reason for not injecting memoization into the code is *No Implementation*. This is expected since many C# constructs (getters, setters, constructors, interfaces) do not contain executable bodies. The third most common reason is also quite understandable. Our approach is designed to store and retrieve values that return from methods and expressions. Therefore, it is logical that methods with

Table 7.4: The number of times retrieval of memoized values caused hits (value found), and misses (value not found) in the memoization storage. An additional column shows the % hit ratio of hits over all the calls to the memoization storage.

project	# Hits	# Misses	Hit Rate [%]
RepoDB	146 491	2 697 404	5.15
Validot	2 172 563	11 331 569	16.09
CsvHelper	493 475	3 003 134	14.10
Enums.NET	43 455 201	10 450 214	**80.61**
Marsen.NetCore.Dojo	16 109	20 705	43.77
MathFlow	10 141 660	7 377 762	**57.90**
Prism	15 670	9 356	**62.62**
sharpPhysics	244 568	17 717	**93.25**
SimplifiedSearch	522 892	153 433	**77.32**
ByteSize	0	0	0.00
StreamingUtilities	407	1 127	26.54
coravel	960	3 281	22.64
NaturalSort.Extension	1 193 956	12 080	**99.00**
PolymorphicJson	0	0	0.00
MemoizationBenchmarking	477	30	**94.08**

a void return type cannot be memoized.

Table 7.5: Total counts of the reasons across all evaluation project to not to inject memoization at compile time, both at method-level and expression-level.

Reason for not Memoizing	Method-level	Expression-level
No Implementation	14 803	0
Extension Method	9 437	0
Void Return Type	8 672	0
Uses Threading Or Asynchronous Operations	3 838	12
Uses External Library Or APIs	2 316	5 922
Alters State Outside Scope	2 040	6
Relies On `this` Reference	1 612	943
Illegal Modifier: `In`	1 403	36
Illegal Modifier: `Out`	1 403	36
Illegal Modifier: `Ref`	605	596
Parent Is Value Type	522	0
Illegal Modifier: `Yield`	263	0

We further examine the limitations caused by parameter modifiers. We have found restricted use of certain parameter modifiers. We use lambda expressions to wrap the computations we memoize. This was done with ease of implementation in mind. However, anonymous functions restrict the use of these keywords. The `yield` keyword is not allowed, because C# does not support anonymous iterator blocks (*CS1621*). In addition, `in`, `out`, `ref` cannot be used inside anonymous functions (*CS1628*). On top of these, we find that about 2000 methods that write to variables outside of its scope. This relates to the variable write-set we discussed in Section 3.2.2. The use of non-local state or API also relates to this. External resources could not be analysed, as such we cannot ensure that there are no side effects or external state changes. Asynchronous operations cannot be memoized because of potentially long execution times and serialization issues. Another limitation we came across during our experiment is the use of structs (`Parent Is Value Type`). Value types such as structs are normally passed by value. It is possible to pass it a reference, using `ref`, however, they cannot be accessed inside lambda bodies

(*CS1673*). Moreover, methods defined inside a struct and extension methods implemented on them also limit memoization for the same reason. A `this` reference to the struct cannot be used inside the lambda.

Because of this limitation, we decided to do a quick analysis on the reasons per project. We found that this limitation on structs is one of the primary reasons memoization could not be implemented in the ByteSize project, see Table 7.6. Most of the data structures defined in the project turn out to be structs with methods, causing them to not be memoized. If another approach instead of using lambda functions were to be used for memoization, the number of memoizations injected at compile time would increase.

Table 7.6: Extracted segment of the reasons memoization did not occur in the ByteSize project. ByteSize shows a frequent constraint on memoization because of limited use of value type structures in lambdas.

Project	Type	Method	Expression
ByteSize	Alters State Outside Scope	12	0
ByteSize	Extension Method	6	0
ByteSize	Illegal ModifiersOut	18	0
ByteSize	No Implementation	12	0
ByteSize	Parent Is Value Type	312	0
ByteSize	Uses External Libraries Or API	12	30
ByteSize	Void Return Type	12	0

7.3 Performance

The third set of metrics we measured concerns the performance of the implementations. We emphasise that the focus of this research is to investigate how well and much memoization can be used, and not to provide an optimal implementation of a memoization approach. We have collected measurements of different steps of the Stryker mutation testing process. These measurements are shown in Table 7.7. The projects in the table are ordered on the lines of code of the projects. The first column with numbers shows the time of the initial test process. Our implementation has made no changes to code regarding this step, yet we still observe a relatively large spread in runtime measurements, often in favour of runner M. The standard deviation of some initial test runs are also high, relative to the time. This indicates that there is a moderate amount of noise that makes it more difficult to accurately observe and determine actual differences in performance. The other columns show the performance measurements of the mutation process, mutant coverage test, the filtering of mutants, and the testing of mutants, in respective order.

The column *Mutate* shows us the time it took Stryker to traverse the syntax trees of the programmes. This is the section of Stryker where the compile time part of our memoization approach is implemented. It is interesting to note that even with the noise we mentioned in the first column, we observe a difference of multiple seconds between mutation processes. Our implementation adds logic to the mutation process and does not remove any. Yet in just five of the fifteen projects, our process was slower than the baseline. It is surprising to see this given the multiple repetitions performed. The mutant coverage test and filtering of mutants showed a slightly smaller variance in measurements compared to the initial test run. These sections have also not been altered by our implementation. The high variation in mutation testing runtimes is notable and may obscure performance comparisons.

The column *Test Mutants* is the most compelling column to observe. This displays the runtime of the total testing process of the mutants, measured from the start of parallel mutation testing to completion of all threads. We observe that projects with longer baseline runtimes tend to experience disproportionately longer runtimes using the memoized version. For short-running projects, the runtime difference

between baseline and memoized versions is minimal. The pattern also seems to somewhat correlate with the size of the projects, including not just the lines of code, but also the number of tests.

Table 7.7: Performance measurements of the primary steps in Stryker's mutation testing process. The results are ordened per proejct and per used runner,

Project	Runner	Initial Test (s)	Mutate (s)	Mutant Coverage (s)	Filter Mutants (s)	Test Mutants (s)
RepoDB.Core	B	2.72 (±0.09)	86.49 (±7.28)	3.39 (±0.38)	0.88 (±0.04)	**282.15**
RepoDB.Core	M	3.84 (±0.73)	78.94 (±33.79)	6.45 (±2.97)	0.93 (±0.05)	**7 527.17**
Validot	B	6.62 (±0.28)	64.42 (±12.80)	8.82 (±0.54)	0.11 (±0.00)	1 448.44
Validot	M	6.10 (±0.33)	67.06 (±15.53)	12.17 (±0.72)	0.12 (±0.00)	2 498.78
CsvHelper	B	2.50 (±0.56)	63.81 (±19.05)	3.60 (±0.10)	0.32 (±0.01)	526.90
CsvHelper	M	2.69 (±0.61)	51.73 (±24.11)	5.43 (±0.56)	0.30 (±0.02)	2 531.64
Enums.NET	B	2.57 (±0.10)	42.00 (±5.68)	3.87 (±1.07)	0.28 (±0.05)	129.28
Enums.NET	M	3.66 (±0.21)	45.70 (±6.98)	4.45 (±1.51)	0.59 (±0.05)	819.90
Marsen.DoJo.Net	B	2.31 (±0.64)	26.77 (±18.16)	1.85 (±0.88)	0.03 (±0.00)	415.92
Marsen.DoJo.Net	M	1.95 (±0.43)	36.68 (±25.95)	2.81 (±0.97)	0.03 (±0.01)	390.57
MathFlow	B	1.31 (±0.19)	40.57 (±4.46)	2.30 (±0.18)	0.22 (±0.01)	285.04
MathFlow	M2	1.08 (±0.03)	26.50 (±4.88)	5.48 (±1.61)	0.21 (±0.02)	1 125.59
Prism.Core	B	4.29 (±0.05)	21.02 (±4.70)	4.51 (±0.04)	0.09 (±0.02)	35.61
Prism.Core	M	4.27 (±0.04)	18.73 (±0.33)	4.68 (±0.08)	0.10 (±0.02)	53.73
SharpPhysics	B	0.69 (±0.15)	20.34 (±9.47)	1.19 (±0.11)	0.11 (±0.03)	45.64
SharpPhysics	M	0.60 (±0.08)	15.09 (±1.60)	1.87 (±0.18)	0.10 (±0.01)	104.66
SimplifiedSearch	B	3.81 (±0.18)	19.52 (±9.74)	3.75 (±3.43)	0.02 (±0.00)	101.06
SimplifiedSearch	M	3.97 (±0.49)	26.82 (±14.26)	2.57 (±0.27)	0.02 (±0.00)	145.89
ByteSize	B	0.78 (±0.21)	12.92 (±9.87)	0.87 (±0.05)	0.06 (±0.00)	28.72
ByteSize	M	0.71 (±0.03)	9.53 (±1.44)	1.16 (±0.11)	0.06 (±0.00)	29.27
StreamingUtilities	B	1.46 (±0.09)	36.21 (±20.66)	1.71 (±0.57)	0.02 (±0.00)	11.86
StreamingUtilities	M	1.41 (±0.07)	23.53 (±1.15)	1.48 (±0.02)	0.02 (±0.01)	10.27
Coravel.Mailer	B	1.04 (±0.05)	25.76 (±18.53)	1.72 (±0.23)	0.02 (±0.00)	**25.62**
Coravel.Mailer	M	1.03 (±0.01)	13.87 (±0.90)	1.49 (±0.05)	0.02 (±0.00)	**27.91**
NaturalSort.Extension	B	0.79 (±0.14)	10.15 (±2.01)	0.67 (±0.02)	0.02 (±0.00)	54.77
NaturalSort.Extension	M	0.73 (±0.04)	8.18 (±0.40)	0.96 (±0.08)	0.02 (±0.00)	146.80
PolymorphicJson	B	0.75 (±0.01)	7.46 (±0.43)	0.69 (±0.01)	0.00 (±0.00)	9.63
PolymorphicJson	M	0.74 (±0.00)	7.27 (±0.20)	0.95 (±0.18)	0.00 (±0.00)	9.58
MemoizationBenchmarking	B	0.78 (±0.11)	11.61 (±3.78)	0.66 (±0.02)	0.01 (±0.00)	46.05
MemoizationBenchmarking	M2	0.84 (±0.21)	14.22 (±8.53)	1.12 (±0.32)	0.01 (±0.00)	45.79

8. Discussion and Conclusion

8.1 Discussion

The general research question guiding this thesis was: How can memoization be used in mutation testing to reduce duplicate computations of non-mutated, shared code?

We started our research by addressing how memoization can be implemented in a programme that is subjected to mutation testing. To explore this, we performed a literature study to understand the known concepts and limitations of mutation testing and memoization. Based on this information, we designed a generalised approach to perform memoization on functions and expressions. This approach can be adopted for any programming language that allows the use of some shared source for the memoization storage.

While this approach established how memoization can be applied in a mutated programme, it raised a practical challenge: determining which parts of the program can be safely memoized without affecting the correctness of mutation testing. This led to our next sub-question: Is it possible to automatically identify code in a (mutated) programme that can be safely memoized?" To examine this, we conducted literature research and practical investigation in C# to compile a list of theoretical and practical constraints a computation must satisfy to be safely memoized. Automating this process proved challenging, Side effects, in particular, limited the number of computations that could be safely memoized. With the use of static analysis methods, we were able to automatically detect the subset of computations that did not suffer from side effects. Our research is pre-emptively classifies a computation as having side-effects without analysing external code that is called that may actually be side-effect free.

Although we were able to automatically identify a subset of computations for statically safe memoization, this approach revealed another challenge: preventing unintended modifications of stateful values stored in the memoization store during mutation testing. Our solution to this problem was straightforward but effective. By using serialisation and deserialisation, we were able to convert programme state into a storable format, allowing copies of the state to be used without risk of modification. While this approach proved feasible, it further restricted the subset of computations that could be safely memoized, as consistency could only be ensured for serialisable types. To address this, we employed a dynamic validation mechanism to verify the serialisability property of computations at runtime.

That said, we still had the obstacle of how to effectively store and share a memoized state between isolated executions of mutated programmes. Each mutation execution runs in its own process, meaning that memoized data cannot be directly shared in memory. Recomputing memoized values for every run would negate the performance benefits of memoization in the repeated execution environment of mutation testing. To address this, we explored data storage mechanisms that could retain memoized data while allowing access across processes. By serialising the memoized values, we were able to store them in a low-level format using memory-mapped files and retrieve them when queried. This approach noticeably reduced redundant computations, although it introduced additional overhead in reading and writing large serialised states. A faster processor would likely not have noticeably reduced this overhead as this mostly has to do with the algorithm locking access to maintain consistency.

8.2 Conclusion

Mutation testing is a computationally expensive process, despite significant performance improvements over recent decades. Motivated by recent research, this thesis investigated how memoization can be applied in mutation testing to reduce duplicate computations of non-mutated code. We explored this through both theoretical analysis and an empirical experiment on 15 real-world projects.

Our findings show that, consistent with prior research, memoization is challenging to fully utilize in complex environments with state and side effects. Across our experiments, roughly 10–56% of computations were statically memoizable, and 4–22% were dynamically memoizable using serialization. We observed that mathematical and string-based libraries performed considerably better on average compared to data model and HTTP libraries. Additionally, half of the projects demonstrated a memoization hit rate exceeding 50%, indicating that memoization can be beneficial in practice.

In our implementation, we focused on lambda functions for convenience and to demonstrate the practical usability of memoization, rather than to optimise performance. However, this choice unintentionally reduced the overall applicability of memoization, since lambda functions in C# impose stricter restrictions on the types of arguments that can be used and passed. Nevertheless, our results demonstrate that memoization can still be viable for mutation testing in certain types of projects, particularly those focused on mathematical computations or string processing.

8.3 Future Work

Although this work has addressed some of the utilizability and effectiveness of memoization for mutation testing, there are several areas that can be further explored. Future research could focus on investigating different limitations of memoization and how to solve them. We have outlined some ideas for future work in the following subsections.

8.3.1 Mutation Testing of Functional Languages

Our work investigates the idea of applying memoization to stateful values. This causes limitations in the ability to memoize. A perhaps more fitting scenario that can be explored is the utilizability of memoization for mutation testing in functional programming languages. Projects in functional languages generally use many pure functions and less functions with side-effects. This may be specifically interesting because side-effects are generally executed in a monadic class that lifts computations out of their pure function into an environment that allows side-effects and IO operations. Research on this topic may reveal interesting memoizability properties of functions that can be used to more optimally and correctly detect the memoizability of functions.

8.3.2 Automated Cost-Benefit Analysis

While this study has investigated the detection of memoizable code in programming languages with stateful values, we have not included an analysis to determine whether memoizing a segment of code is likely to improve the performance. Future work could extend our work on automatic detection of memoizable code in these languages to include a computational cost-benefit analysis to verify whether we can automatically determine if a segment of code is worthwhile to memoize. Test-coverage information, test-profiling information, and static analysis might be able to be combined to determine the relative gain expected when memoizing code. Such advances could improve the practical effectiveness

of memoization for mutation testing by reducing the overhead of memoizing code that is not expected to yield a performance gain.

8.3.3 Shared Managed Memory Space

Although we used Memory-Mapped Files because they are widely supported by operating systems, they also showed us that unmanaged memory has its practical limitations. We cannot directly share data between processes and need to manage its conversion and concurrent access at the byte-level. A study could be done to investigate whether it is possible to share or combine managed memory spaces of (different instances of) the same process. Additionally, exploring the concept of combined or distributed garbage collectors could pioneer research in the dynamic usage of managed memory spaces. If such approaches are proved to be feasible, then sharing memoized values across mutation testing processes may be possible in a more effective and direct way.

Bibliography

Abuljadayel, A., & Fadi Wedyan. (2018). An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms. *International Journal of Intelligent Systems and Applications*, *10*(1), 34–45. https://doi.org/10.5815/ijisa.2018.01.05

Acar, U. A., Blelloch, G. E., & Harper, R. (2003). Selective memoization.

Acar, U. A., Blelloch, G. E., & Harper, R. (2004, November). *Adaptive Memoization*.

Ahmed, Z., Zahoor, M., & Younas, I. (2010). Mutation operators for object-oriented systems: A survey. *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, *2*, 614–618. https://doi.org/10.1109/ICCAE.2010.5451692

Ausiello, G., Demetrescu, C., Finocchi, I., & Firmani, D. (2012). K-Calling context profiling. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 867–878. https://doi.org/10.1145/2384616.2384679

Besnard, L., Pinto, P., Lasri, I., Bispo, J., Rohou, E., & Cardoso, J. M. (2019). A framework for automatic and parameterizable memoization. *SoftwareX*, *10*, 100322. https://doi.org/10.1016/j.softx.2019.100322

Beyer, D., & Friedberger, K. (2020). Domain-independent interprocedural program analysis using block-abstraction memoization. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 50–62. https://doi.org/10.1145/3368089.3409718

Boyer, R. S., & Hunt, W. A. (2006). Function memoization and unique object representation for ACL2 functions. *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications - ACL2 '06*, 81. https://doi.org/10.1145/1217975.1217992

Budd, T. A., & Angluin, D. (1982). Two notions of correctness and their relation to testing [Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer-Verlag]. *Acta Informatica*, *18*(1), 31–45. https://doi.org/10.1007/BF00625279

Cañizares, P. C., Núñez, A., Filgueira, R., & de Lara, J. (2024). Parallel mutation testing for large scale systems [Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 2 Publisher: Springer US]. *Cluster Computing*, *27*(2), 2071–2097. https://doi.org/10.1007/s10586-023-04074-y

Cardoso, J. M. P., Coutinho, J. G. F., Carvalho, T., Diniz, P. C., Petrov, Z., Luk, W., & Gonçalves, F. (2016). Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach. *Software: Practice and Experience*, *46*(2), 251–287. https://doi.org/10.1002/spe.2301

Choi, B., Mathur, A., & Pattison, B. (1989). PMothra: Scheduling mutants for execution on a hypercube. *SIGSOFT Softw. Eng. Notes*, *14*(8), 58–65. https://doi.org/10.1145/75309.75316

Choi, B., DeMillo, R., Krauser, E., Martin, R., Mathur, A., Offutt, A., Pan, H., & Spafford, E. (1989). The Mothra tool set (software testing). *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, *2*, 275–284 vol.2. https://doi.org/10.1109/HICSS.1989.48002

Coles, H. (2015). PIT Mutation Testing. Retrieved April 2, 2025, from https://pitest.org/

Coviello, C., Romano, S., Scanniello, G., Marchetto, A., Corazza, A., & Antoniol, G. (2020). Adequate vs. inadequate test suite reduction approaches. *Information and Software Technology*, *119*, 106224. https://doi.org/10.1016/j.infsof.2019.106224

Davis, J. C., Servant, F., & Lee, D. (2021). Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS) [ISSN: 2375-1207]. *2021 IEEE Symposium on Security and Privacy (SP)*, 1–17. https://doi.org/10.1109/SP40001.2021.00032

de Roos, M., Rensink, A., & van Hees, R. (2024). Faster Mutation Testing through Simultaneous Mutation Testing.

DeMillo, R., Lipton, R., & Sayward, F. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, *11*(4), 34–41. https://doi.org/10.1109/C-M.1978.218136

Fernandes, L., Ribeiro, M., Carvalho, L., Gheyi, R., Mongiovi, M., Santos, A., Cavalcanti, A., Ferrari, F., & Maldonado, J. C. (2017). Avoiding useless mutants. *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 187–198. https://doi.org/10.1145/3136040.3136053

Field, A., & Harrison, P. (1988, January). *Functional programming*.

Finifter, M., Mettler, A., Sastry, N., & Wagner, D. (2008). Verifiable functional purity in java. *Proceedings of the 15th ACM conference on Computer and communications security*, 161–174. https://doi.org/10.1145/1455770.1455793

Fornace, M. E., Porubsky, N. J., & Pierce, N. A. (2020). A Unified Dynamic Programming Framework for the Analysis of Interacting Nucleic Acid Strands: Enhanced Models, Scalability, and Speed. *ACS Synthetic Biology*, *9*(10), 2665–2678. https://doi.org/10.1021/acssynbio.9b00523

Fraser, G., Weiglhofer, M., & Wotawa, F. (2008). Coverage Based Testing with Test Purposes [ISSN: 2332-662X]. *2008 The Eighth International Conference on Quality Software*, 199–208. https://doi.org/10.1109/QSIC.2008.41

Friedberger, K. (2015, November). *Block-Abstraction Memoization as an Approach to Verify Recursive Procedures* [Master's thesis, University of Passau]. https://www.sosy-lab.org/research/msc/2015.Friedberger.Block-Abstraction_Memoization_as_an_Approach_to_Verify_Recursive_Procedures.pdf

Friedberger, K. (2021, May). *Efficient Software Model Checking with Block-Abstraction Memoization* [PhD Dissertation]. Ludwig Maximilian University of Munich. https://doi.org/10.5282/edoc.29976

Fujinami, H., & Hasuo, I. (2024). Efficient Matching with Memoization for Regexes with Look-around and Atomic Grouping [ISSN: 1611-3349]. *Programming Languages and Systems*, 90–118. https://doi.org/10.1007/978-3-031-57267-8_4

Ghanbari, A., & Marcus, A. (2022). Faster mutation analysis with MeMu. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 781–784. https://doi.org/10.1145/3533767.3543288

Gopinath, R., Jensen, C., & Groce, A. (2016). Topsy-Turvy: A smarter and faster parallelization of mutation analysis. *Proceedings of the 38th International Conference on Software Engineering Companion*, 740–743. https://doi.org/10.1145/2889160.2892655

IEEE, The Open Group. (2018). Mmap Specifications. Retrieved October 8, 2025, from https://pubs.opengroup.org/onlinepubs/9699919799/

Info Support. (2025). Stryker Mutator. Retrieved March 10, 2025, from https://stryker-mutator.io/

Jehan, S., & Wotawa, F. (2023). An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage. *IEEE Access*, *11*, 65427–65442. https://doi.org/10.1109/ACCESS.2023.3289073

Jia, Y., & Harman, M. (2008). MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. *Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)*, 94–98. https://doi.org/10.1109/TAIC-PART.2008.18
Mentions Higher Order mutation testing, but not first to propose it.

Jia, Y., & Harman, M. (2009). Higher Order Mutation Testing. *Information and Software Technology*, *51*(10), 1379–1393. https://doi.org/10.1016/j.infsof.2009.04.016

Jia, Y., & Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing [Conference Name: IEEE Transactions on Software Engineering]. *IEEE Transactions on Software Engineering*, *37*(5), 649–678. https://doi.org/10.1109/TSE.2010.62

Jin, Y., Wang, H., Yu, T., Tang, X., Hoefler, T., Liu, X., & Zhai, J. (2020). SCALANA: Automating Scaling Loss Detection with Graph Analysis. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–14. https://doi.org/10.1109/SC41405.2020.00032

Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., & Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing? *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 654–665. https://doi.org/10.1145/2635868.2635929

Kedia, P., Costa, M., Parkinson, M., Vaswani, K., Vytiniotis, D., & Blankstein, A. (2017). Simple, fast, and safe manual memory management. *ACM SIGPLAN Notices*, *52*(6), 233–247. https://doi.org/10.1145/3140587.3062376

Langdon, W. B., Harman, M., & Jia, Y. (2010). Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, *83*(12), 2416–2430. https://doi.org/10.1016/j.jss.2010.07.027

Ma, Y.-S., Kwon, Y.-R., & Offutt, J. (2002). Inter-class mutation operators for Java [ISSN: 1071-9458]. *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, 352–363. https://doi.org/10.1109/ISSRE.2002.1173287

Ma, Y.-S., Offutt, J., & Kwon, Y. R. (2005). MuJava: An automated class mutation system. *Software Testing, Verification and Reliability*, *15*(2), 97–133. https://doi.org/10.1002/stvr.308

Madeyski, L., Orzeszyna, W., Torkar, R., & Józala, M. (2014). Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation [Conference Name: IEEE Transactions on Software Engineering]. *IEEE Transactions on Software Engineering*, *40*(1), 23–42. https://doi.org/10.1109/TSE.2013.44

Marsit, I., Ayad, A., Kim, D., Latif, M., Loh, J., Omri, M. N., & Mili, A. (2021). The ratio of equivalent mutants: A key to analyzing mutation equivalence. *Journal of Systems and Software*, *181*, 111039. https://doi.org/10.1016/j.jss.2021.111039

Michie. (1968). "Memo" Functions and Machine Learning. *Nature Publishing Group UK London*, *218*, 19–22. https://doi.org/10.1038/218019a0

Microsoft. (2022, July). CreateFileMappingA function (winbase.h) - Win32 apps. Retrieved October 8, 2025, from https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga

Mustard, C., & Fedorova, A. (2018). Practical Cross Program Memoization with KeyChain. *2018 IEEE International Conference on Big Data (Big Data)*, 262–271. https://doi.org/10.1109/BigData.2018.8622210

Nistor, A., Song, L., Marinov, D., & Lu, S. (2013). Toddler: Detecting performance problems via similar memory-access patterns [ISSN: 1558-1225]. *2013 35th International Conference on Software Engineering (ICSE)*, 562–571. https://doi.org/10.1109/ICSE.2013.6606602

Norvig, P. (1991). Technical Correspondence Techniques for Automatic Memoization with Applications to Context-Free Parsing. *17*(1).

Offutt, A. J., & Untch, R. H. (2001). Mutation 2000: Uniting the Orthogonal. Retrieved March 10, 2025, from https://www.albany.edu/faculty/offutt/research/papers/mut00.pdf

Offutt, J., Ma, Y.-S., & Kwon, Y.-R. (2006). The class-level mutants of MuJava. *Proceedings of the 2006 international workshop on Automation of software test*, 78–84. https://doi.org/10.1145/1138929.1138945

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., & Harman, M. (2019, January). Chapter Six - Mutation Testing Advances: An Analysis and Survey. In A. M. Memon (Ed.), *Advances in Computers* (pp. 275–378, Vol. 112). Elsevier. https://doi.org/10.1016/bs.adcom.2018.03.015

Pizzoleto, A. V., Ferrari, F. C., Offutt, J., Fernandes, L., & Ribeiro, M. (2019). A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, *157*, 110388. https://doi.org/10.1016/j.jss.2019.07.100

Raposo, C., & Mago Quintao Pereira, F. (2024, April). Memoization of Mutable Objects. Retrieved March 7, 2025, from https://sol.sbc.org.br/index.php/sblp/article/view/30251/30058

Rito, H., & Cachopo, J. (2010). Memoization of methods using software transactional memory to track internal state dependencies. *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, 89–98. https://doi.org/10.1145/1852761.1852775

Shirazi, B., Wang, M., & Pathak, G. (1990). Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, *10*(3), 222–232. https://doi.org/10.1016/0743-7315(90)90014-G

Silva, R. A., Senger de Souza, S. d. R., & Lopes de Souza, P. S. (2017). A systematic review on search based mutation testing. *Information and Software Technology*, *81*, 19–35. https://doi.org/10.1016/j.infsof.2016.01.017

Stoffers, M., Schemmel, D., Dustmann, O. S., & Wehrle, K. (2018). On Automated Memoization in the Field of Simulation Parameter Studies. *ACM Transactions on Modeling and Computer Simulation*, *28*(4), 1–25. https://doi.org/10.1145/3186316

Stoffers, M., Schemmel, D., Soria Dustmann, O., & Wehrle, K. (2016). Automated Memoization for Parameter Studies Implemented in Impure Languages. *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 221–232. https://doi.org/10.1145/2901378.2901386

Sun, Y., Peng, X., & Xiong, Y. (2023). Synthesizing Efficient Memoization Algorithms. *Artifact for "Synthesizing Efficient Memoization Algorithms"*, *7*(OOPSLA2), 225:89–225:115. https://doi.org/10.1145/3622800

Suresh, A., Rohou, E., & Seznec, A. (2017). Compile-time function memoization. *Proceedings of the 26th International Conference on Compiler Construction*, 45–54. https://doi.org/10.1145/3033019.3033024

Untch, R. H., Offutt, A. J., & Harrold, M. J. (1993). Mutation analysis using mutant schemata. *Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, 139–148. https://doi.org/10.1145/154183.154265

Usaola, M. P., & Mateo, P. R. (2010). Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software*, *27*(3), 80–86. https://doi.org/10.1109/MS.2010.79

Vercammen, S., Demeyer, S., Borg, M., & Eldh, S. (2018). Speeding up Mutation Testing via the Cloud:Lessons Learned for Further Optimisations.

Walkinshaw, N., & Minku, L. (2018). Are 20% of files responsible for 80% of defects? *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–10. https://doi.org/10.1145/3239235.3239244

Wanninger, N., McMichen, T., Campanoni, S., & Dinda, P. (2024). Getting a Handle on Unmanaged Memory. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 448–463. https://doi.org/10.1145/3620666.3651326

Wong, C.-P., Meinicke, J., Chen, L., Diniz, J. P., Kästner, C., & Figueiredo, E. (2020). Efficiently finding higher-order mutants. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1165–1177. https://doi.org/10.1145/3368089.3409713

Wonisch, D. (2012). Block Abstraction Memoization for CPAchecker [ISSN: 1611-3349]. *Tools and Algorithms for the Construction and Analysis of Systems*, 531–533. https://doi.org/10.1007/978-3-642-28756-5_41

Wonisch, D., & Wehrheim, H. (2012). Predicate Analysis with Block-Abstraction Memoization [ISSN: 1611-3349]. *Formal Methods and Software Engineering*, 332–347. https://doi.org/10.1007/978-3-642-34281-3_24