TU WIEN Informatics

Improving Rust Mutation Testing using Static and Dynamic Program Analysis

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Samuel Pilz, BSc.
Registration Number 1327391

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Jens Knoop

Vienna, 13th September, 2023

Samuel Pilz	Jens Knoop

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Samuel Pilz, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. September 2023

Samuel Pilz

Acknowledgements

I want to thank my supervisor Univ.Prof. Jens Knoop for his valuable support and thoughtful advice. I am very thankful to Hannes Siebenhandl and Jana Chadt for their constructive and insightful discussions and feedback. I am especially grateful for the support from my wife throughout my studies.

Abstract

Mutation testing is a powerful software testing method in which the program under test is seeded with artificial faults that are considered to be possible programming errors and should be discovered by a high-quality test suite. The cost of mutation testing is one of the most critical issues for its practical applications. In this thesis, we present the tool `muttest` for mutation analysis of Rust programs that improves quality and performance of mutation analysis compared to related systems.

This thesis proposes several methods for leveraging Rust language features and results of static analysis to implement mutation operators while preventing the generation of invalid mutations. Moreover, dynamic program analysis is used to perform weak mutation analysis and its results are utilized to improve the performance of strong mutation analysis. By relying only on stable features of Rust, we ensure best-possible compatibility of `muttest` with future versions of Rust. The experimental evaluation in this thesis shows that `muttest` has competitive performance and produces a high-quality mutation analysis report.

Kurzfassung

Mutationstesten ist eine mächtiger Prozess um Software zu testen. Dabei wird das Zielprogramm mit künstlichen Fehlern versehen, die als mögliche Programmierfehler angenommen werden und die durch qualitativ hochwertige Testfälle entdeckt werden sollten. Die Kosten des Mutationstestens sind einer der kritischsten Eigenschaften, welche die praktische Anwendung erschwert. In dieser Arbeit stellen wir das Tool `muttest` für Mutationsanalyse von Rust Programmen vor, das die Qualität und Performanz von Mutationsanalyse von vergleichbaren Systemen übertrifft.

Diese Arbeit stellt einige Techniken vor wie Sprachfeatures von Rust sowie die Ergebnisse von statischer Programmanalyse dazu verwendet werden Mutationsoperatoren zu implementieren, die keine ungültigen Mutationen generieren. Außerdem wird dynamische Programmanalyse dafür verwendet schwache Mutationsanalyse durchzuführen. Diese Ergebnisse werden wiederum dafür eingesetzt die Laufzeit von starker Mutationsanalyse zu verbessern. Durch die Einschränkung auf stabile Sprachfeatures von Rust, stellen wir optimale Kompatibilität von `muttest` mit zukünftigen Versionen von Rust sicher. Die experimentelle Evaluation dieser Arbeit zeigt, dass `muttest` konkurrenzfähige Laufzeit aufweist und einen qualitativ hochwertigen Mutationsanalysebericht liefert.

Contents

CHAPTER 1

Introduction

Ensuring correctness of programs is of paramount importance in software engineering. In practice, unit and integration tests, usually bundled in test suites, are often used to establish trust in the correctness of software. For a high level of confidence, the tests need to be well-suited to reveal potential programming errors. Hence, the test suite needs to be constructed using a systematic testing method and its quality needs to be analyzed [AO17].

One method of determining the adequacy of test cases is *coverage-based* analysis [AO17, chap 7]. Listing 1.1 shows a faulty implementation for computing Fibonacci numbers, in which the two previous Fibonacci numbers are subtracted instead of added. The two provided test cases execute all statements and branches of the function `fibs` and would therefore satisfy these coverage testing criteria. However, due to insufficient assertions of the computed values, the test cases do not detect the programming error. The testing method *mutation testing* [DLS78] aims to remedy this weakness.

```
fn fibs(n: i32) -> i32 {
    if n <= 1 {
      return n;
    }
    return fibs(n-1)
         - fibs(n-2);
}        // programming error!
```

```
#[test]
fn fibs_tests() {
  assert!(fibs(2) == 1);
  assert!(fibs(8) >= 1);
}
```

Listing 1.1: A faulty function for computing Fibonacci numbers and corresponding test cases.

1.1 Mutation Testing: Theory and Process

Informally, mutations are small program modifications whose presence should be detected by running the test suite. Mutation testing assumes that altering the program by introducing *mutations* modifies the semantics like a programming error would have done. When running the test suite against one of the resulting *mutants*, the violation of an assertion means that the test suite is able to catch the modification of the program and the mutant is *killed*. However, if all tests pass when run against the altered program, the mutant is called *undetected* or *survived* and the corresponding programming error would remain hidden from the test suite, hinting at a possible test suite weakness. If the test suite does not even execute the altered part of the code, the mutation is *not covered* and therefore the mutant cannot be killed. A *kill matrix* tabulates which test case covers and kills which mutant.

Mutants that cause the test suite to run indefinitely are considered killed. Obviously, proving nontermination is not possible, we therefore assume that a test will not terminate if it exceeds a certain threshold. The underlying assumption is that test cases are designed such that they terminate within a reasonable time frame and a long-running test suite would raise the suspicion of the tester, if executed manually.

Some mutations do not represent possible programming errors. Mutants that fail to compile are called *invalid*, whereas *equivalent* mutations do not alter the program's observable behavior, making them impossible to detect. The quality of a test suite can be measured in terms of its *mutation score*, which is the ratio of killed mutants compared to the number of valid, non-equivalent mutants. Automated tools, however, cannot determine the equivalence of mutations reliably and pessimistically count all valid mutations as non-equivalent, underestimating the quality of the test suite.

Mutants are typically generated via *mutation operators* [Pap+19, p. 7], which are each responsible for altering a distinct pattern of code into possible alternatives. For some operators, the program text contains all information that is necessary to determine the set of valid mutations for a given fragment of code, for example, editing the content of a string literal is a valid change in most programming languages. However, more complicated mutations might depend on global program properties in order to be valid, such as types of variables, whether operators are overloaded for given types, control or data flow, and signatures of defined functions. We call such operators *context sensitive*.

The process of strengthening the test suite with the goal to improve the mutation score can help uncover programming errors. By construction, a test suite with high mutation score detects the simple programming errors that are examined during mutation analysis. DeMillo, Lipton, and Sayward [DLS78] formulate the *coupling effect* that claims that tests which uncover these simple errors typically also uncover more complex errors that are more common in practice. Acree et al. [Acr+79] show that a high mutation score necessarily implies a high coverage score, indicating that mutation testing can lead to stronger test suites compared to coverage-based methods. Andrews, Briand, and Labiche

[ABL05] study the accuracy of mutation scores as a measure of quality empirically and conclude that the mutation score is, in fact, an adequate measure for test suite quality.

We follow the terminology introduced by Amalfitano et al. [Ama+22], which separates the two concepts of *mutation analysis* and *mutation testing*. Mutation analysis is the process of determining, generating, and evaluating mutants, as well as computing the mutation score for a given test suite. In contrast, mutation testing is the act of strategically designing test cases and improving the test suite based on results of mutation analysis.

1.2 The Cost of Mutation Testing

In contrast to other testing methods, like coverage-based approaches, mutation testing is significantly more resource intensive [Woo93]. In particular, Jia and Harman [JH11] argue that the cost of mutation testing in practice is largely determined by

- mutation generation and mutant evaluation,
- manual review of mutation analysis reports, and
- the process of handling mutations that do not represent real programming errors, especially equivalent mutations.

Several methods have been proposed to lower these costs, of which we want to highlight two: mutant schema generation and weak mutation analysis.

Untch, Offutt, and Harrold [UOH93] have developed the method of *Mutant Schema Generation*, also called mutation *baking*, to reduce costs of mutant evaluation for compiled languages. Instead of compiling each mutant separately, the source code is transformed at compile-time to introduce the possibility to alter the program's behavior at run-time. The compiled artifact can be configured at run-time to behave like any mutant, requiring only a single compiler run for when performing mutation analysis.

Howden [How82] introduces *weak mutation analysis*, in which mutants are considered killed if the mutated expression evaluates to a different value or produces different side effects to the original one for some test case. The term *strong mutation analysis* is often used to describe the original approach to emphasize its distinction to weak mutation analysis. Notably, mutants can be killed by weak mutation analysis but survive strong mutation analysis. Therefore, the weak mutation analysis score over-estimates the test suite's ability to detect mutations. However, a single run of the test suite is sufficient to determine which mutants are killed by weak mutation analysis, making this approach an efficient alternative to strong mutation analysis with reduced accuracy. Moreover, mutants that survive weak mutation analysis cannot cause a test assertion to fail making it impossible for these mutants to be killed by strong mutation analysis. As a consequence, there is no necessity for executing the test suite to evaluate these mutants for strong mutation analysis.

```rust
fn min_length_str<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() <= b.len() { a } else { b }
}

// lets the mutant survive strong and weak mutation analysis
assert!(min_length("a", "bc") == "a");

// kills the mutant in weak mutation analysis, but
// lets it survive strong mutation analysis
assert!(min_length("ab", "bc").len() == 2);

// kills the mutant in strong and weak mutation analysis
assert!(min_length("ab", "bc") == "ab");
```

Listing 1.2: We consider the mutation of `<=` into `<` in the following function. The test cases give all possible results for strong and weak mutation analysis.

Listing 1.2 shows an example function and three test cases. We consider the mutation of the comparison operator `<=` into `<`. Dynamic program analysis allows us to determine that both the original and mutated programs behave identically when executing the first test case because the operators `<=` and `<` evaluate to the same result unless the compared values are equal. This means that this test case lets the mutant survive weak mutation analysis. As a consequence, the test case cannot kill the mutant in strong mutation analysis because the mutated function produces the same result and the assertion is not violated. Since this mutation is not equivalent, this finding uncovers a test suite weakness without the cost of strong mutation analysis.

The second test case illustrates the shortcomings of weak mutation analysis. Dynamic program analysis determines that the program executions of the original and mutated differ on the given input. However, because the assertion is poor, the mutant would not cause an assertion error in this test case. Therefore, the test kills the mutant in weak mutation analysis, but lets it survive strong mutation analysis. The third test case is adequate to kill the mutant in strong and weak mutation analysis.

1.3 Evolution and State of the Art: Mutation Analysis Tools

Since its introduction by DeMillo, Lipton, and Sayward [DLS78] and Acree et al. [Acr+79], mutation analysis and mutation testing have been a active areas of research; multiple surveys have summarized and reviewed the literature at different points in time [Woo93; OU01; JH11; Pap+19]. Many tools for mutation analysis have been developed for several programming languages. These feature a variety of methods to implement the evaluation of mutants. An overview of selected mutation analysis tools for different programming

Table 1.3: Comparison of existing mutation analysis tools

Tool	Year	Language	Mutation method	Program analysis
unnamed [Bud+78]	1978	Fortran	Source code edit	Syntax patterns only
unnamed [How82]	1982	COBOL	Weak only	Syntax patterns & Dynamic
`mothra` [KO91]	1991	Fortran	Bytecode (run-time)	Syntax patterns only
MSG [UOH93]	1993	Fortran	Baked	Syntax patterns only
`mutandis` [MMP13]	2013	Javascript	Instrumentation	Static & Dynamic
`Major` [Jus14]	2014	Java	Baked	Static
`pit` [Col+16]	2014	Java	Bytecode (run-time)	Bytecode
`MutPy` [DH14]	2014	Python	Source code edit	Syntax patterns only
`MuCheck` [Le+14]	2014	Haskell	Source code edit	Types
`mutagen` [Bog18]	2018	Rust	Baked	Syntax patterns only
`Mull` [DP18]	2018	C/ C++	Bytecode (compile-time)	Bytecode
`cargo-mutants` [sou23]	2021	Rust	Source code edit	Static
`mutest-rs` [Lév22]	2022	Rust	Baked	Static
`muttest` (this thesis)	2023	Rust	Baked	Static & Dynamic

languages and their methods for mutant generation is shown in Table 1.3. A more exhaustive collection is compiled by Papadakis et al. [Pap+19].

The set of valid mutations is highly dependent on the programming language's syntax and semantics, especially its type system. Moreover, each language has unique constraints and possibilities on how mutation generation and evaluation can be implemented. Due to these engineering challenges, most of the developed systems are language-specific. Early systems generate mutants by editing the source code directly. The altered program is then compiled from scratch or loaded into a fresh interpreter session. Using this approach, Budd et al. [Bud+78] implement their system for the programming language Fortran.

Most mutation analysis tools focus on simple mutation operators built on pattern-based analysis of program text alone. Notably, in the language Fortran mutations regarding arithmetic and control flow, which are the most popular ones, can be generated this way [Bud+78]. However, in some high-level programming languages, some operations are overloaded, making the corresponding operators context sensitive. For example, in Java, Python, or JavaScript, the plus (`+`) operator can be used to append strings and replacing

it with a minus (-) is not a valid mutation. In order to support this mutation operator correctly, a mutation analysis tool requires type information, either by performing static program analysis, or by inspecting the compiler-generated bytecode.

Several tools have been developed to explore cost-reduction techniques for mutation testing. Howden [How82] use dynamic program analysis to implement weak mutation analysis. Untch, Offutt, and Harrold [UOH93] propose to implement mutation analysis tools based on baking mutations using compile-time source code transformations, reducing cost of compiling mutants. The framework `mutandis` [MMP13] uses static and dynamic program analysis to rank mutations by estimated importance and evaluate only the most important ones.

The system `pit` [Col+16] exploits the functionality of the Java Virtual machine [AGH05] to edit bytecode at run-time. Mutants are evaluated by performing bytecode changes within the same process, removing the startup costs for test suites. The tool `mull` [DP18] evaluates mutants by editing LLVM bytecode, which only requires recompilation of minimal pieces of code. The approach to use a low-level representation of the program for mutation analysis seems particularly useful when targeting programming languages with complex syntax, like C and C++, since possible valid mutations cannot be easily determined from analyzing the source code [JP17].

Mutation analysis tools that operate solely on bytecode can provide invalid mutations, as shown by the [Jus14]. They give the example of for-each loops that have identical bytecode to semantically equivalent while loops. A mutation analysis tool without knowledge of the source code might incorrectly apply mutations for while loops, which would be invalid when applied to for-each loops. To address this weakness, the authors present the mutation analysis tool `Major` based on mutation baking.

The mutation analysis tools `mutagen` [Bog18], `mutest-rs` [Lév22], and `cargo-mutants` [sou23] have been developed for Rust. In `mutagen` and `mutest-rs`, mutations are baked at compile-time, while `cargo-mutants` edits the source code and calls the compiler separately for each mutant. The systems `mutagen` and `mutest-rs` support traditional mutation operators [Acr+79], altering the behavior of expressions and statements. In contrast, `cargo-mutants` only applies "extreme" mutation operators [NJW16], removing whole function bodies or replacing them by trivial values, which improves performance and simplifies manual review, but also reduces the ability to find programming errors. The source code transformation of `mutagen` is implemented as a procedural macro, while `mutest-rs` provides a compiler plugin that bakes mutations. Contrary to `cargo-mutagen` and `mutest-rs`, which execute the test suite for each mutant, `mutagen` instruments the code to detect and skip the evaluation of mutants that are not covered.

Both tools support type-sensitive mutation operators whose validity depend on trait implementations. The strategy of multiple compiler invocations allows `cargo-mutants` to filter these mutations and correctly exclude these for mutation score computation. In `mutagen`, however, mutants are generated based on patterns of program text alone and invalid mutations fail at run-time, skewing the mutation score. Neither system can be

```rust
#[proc_macro_attribute]
fn my_macro(attribute: TokenStream, code: TokenStream)
  -> TokenStream {
  // arbitrary program to define the transformation
}
```

Listing 1.4: A Rust procedural macro for compile-time source code transformation

used to perform weak mutation analysis [How82].

The accuracy of the mutation score as measure for test suite quality critically depends on the set of mutation operators. Previous research has focussed on minimizing the set of mutation operators while minimizing the loss of measurement accuracy, finding that a small subset of possible mutations can provide an adequate estimate for test suite quality [Off+96; NAM08]. In their study, Andrews, Briand, and Labiche [ABL05] examine operators for arithmetic operations, constants, branches, and statements and conclude that these operators seem sufficient for mutation analysis for the C programming language.

1.4 The Rust Programming Language

Rust [KN19] is a systems programming language with a type system designed to prevent invalid memory access, such as data races and the use of dangling references. This is achieved by extending its type system with the concepts of ownership and borrowing. Informally, a valid borrow of a value guarantees that a value or resource can be used safely while ownership ensures that each resource is freed appropriately and cannot be used after that. Rust follows a procedural paradigm heavily influenced by functional programming languages. Rust supports parametric polymorphism via so-called traits, which are similar to Haskell's type classes. Projects written in Rust are organized in workspaces, which may contain multiple libraries, called *crates*. The tool `cargo` is used for building Rust projects and managing their dependencies.

Like any programming language with an active community, Rust is evolving. Experimental features can be used by installing a beta version of the compiler. Unlike stable features, these are still subject to further change and projects relying on them cannot be expected to be compatible with future versions of Rust.

As a core part of the language, Rust allows procedural macros to generate and transform the source code at compile-time. They are implemented as Rust programs that take source code as input and produce source code as output. Listing 1.4 shows the signature. When annotating code with the appropriate attribute, like `#[my_macro]`, the procedural macro is invoked by the compiler to transform the annotated code fragment. This happens at an early compilation stage in which the compiler has not yet performed semantic analysis. Therefore, procedural macros do not have access to the type information of the code under transformation.

```rust
trait Print {
    fn print(&self);
}
struct A;
impl Print for A {
    fn print(&self) {
        print!("A")
    }
}
impl Print for &A {
    fn print(&self) {
        print!("&A")
    }
}
```

```rust
struct B;
impl Print for B {
    fn print(&self) {
        print!("B")
    }
}

(&A).print(); // prints "A"
(&&A).print(); // prints "&A"
(&B).print(); // prints "B"
(&&B).print(); // prints "B"
```

Listing 1.5: Examples of method resolution rules. The effect of the expression `(&&B).print()` is identical to that of `(&B).print()` while `(&&A).print()` and (&A).print() behave differently because the trait `Print` is implemented for both `A` and `&A`

1.4.1 Autoderef-based Specialization

Rust performs multiple steps to resolve calls to trait methods that take `&self` as a parameter. First, the trait is resolved for the type of expression. If no method is found and the value can be dereferenced, the compiler tries to resolve the method based on the dereferenced form of the value. This process is repeated until a matching method is found or the value can no longer be dereferenced. This step is called *autoderef* and was introduced to allow syntax for method calls that is similar to related concepts in popular object-oriented programming languages. Listing 1.5 illustrates this concept.

Tolnay [Tol] outlines how the related feature of autoref-based method resolution is used to implement compile-time specialization in which different implementations are chosen based on the type of an expression. This technique is refined by Kalbertodt [Kal] to use autoderef-based method resolution and allows multiple layers of specialization. Notably, this approach does not rely on experimental features and is therefore expected to be compatible with future versions of Rust.

1.5 Contributions of this Thesis

In this thesis, we develop a new system for mutation analysis in Rust called `muttest`. We build on top of the method of baking mutations through source code transformation, augmenting the performance and quality of mutation analysis of existing systems by integrating static and dynamic program analysis.

We introduce the concept of a *mutable*, a new abstraction for implementing mutation analysis systems. A mutable is a code fragment whose behavior can be altered. Using

this definition, we improve the modularity of the implementation of `muttest` compared to related systems.

We develop a novel mechanism that leverages the compiler itself to perform all static analyses required by the mutation operators supported by `muttest`. This implies that our approach remains applicable to all Rust projects compatible with an up-to-date version of the Rust compiler. The stability guarantees of Rust give us confidence that `muttest` will mostly be compatible with future versions of the compiler without much need for adaption.

We develop a new strategy to apply the Rust pattern *autoderef-based specialization* to implement context sensitive mutation operators in a way that they never generate invalid mutations. This allows us to define mutation operators that are either not possible or not practical in related systems. Since most mutation operators of complex language constructs are context sensitive, this enables particularly powerful mutation operators. Eliminating the generation of invalid mutations yields the additional benefit of removing this source of over-approximation of the mutation score.

We bake instrumentation alongside each mutation that records if and how each test case covers the mutation. We extend the strategies of existing tools by also recording the behavior of certain code fragments in addition to coverage. This dynamic analysis data allows us to skip test cases which are not relevant for a given mutation, either by not covering it or by letting the mutation survive weak mutation analysis. This gives a significant performance boost, since most unit tests only cover a fraction of the overall code base.

We leverage the data gathered from instrumentation to enable weak mutation analysis for mutation operators where possible. The weak mutation score can be computed during a single execution of the test suite, which allows the programmer to quickly compute an upper bound to the strong mutation analysis score. We expect this is a more efficient strategy in practice to construct test cases that kill most mutants by weak mutation analysis before studying the more detailed report from strong mutation analysis.

We compare `muttest` to other mutation analysis systems for Rust programs. In order to get results that are comparable with other studies, we use small example programs that have been widely used in previous research. A benchmark consisting of several well-known libraries in the Rust ecosystem shows that `mutest` generates more mutants than related systems and is as efficient as the best-performing systems.

We follow software engineering best-practices to test `muttest` itself to establish confidence in its correctness. In particular, a carefully crafted implementation of core mutation features enables isolated unit tests. Furthermore, we apply `muttest` to itself and ensure a reasonably high mutation score.

In the remaining chapters of this thesis, we present our work on the system `muttest` as outlined above. In Chapter 2 we discuss its high-level design concept and the relevant components. We give implementation details of the key ideas in Chapter 3 and evaluate

the system in Chapter 4. Finally, we discuss the results of our work and give an outlook for possible future research in Chapter 5.

CHAPTER 2

Design and Architecture of the Rust Mutation Analysis Tool `muttest`

We present the design and architecture of the system `muttest` for mutation analysis of Rust programs. We first discuss its requirements which form the basis for the following design decisions. Then, we describe the steps `muttest` executes to generate and evaluate mutants and give an overview of how `muttest` is used by programmers to perform mutation testing. Thereafter, we present the component structure of the `muttest` project and outline the structure of the procedural macros that implement the source code transformations. Finally, we discuss how `muttest` is tested and how it is used on its own test suite to compute the mutation score.

2.1 Requirements for `muttest`

We identify the following major requirements that the mutation analysis tool `muttest` should meet. These requirements are listed by decreasing importance. Design decisions have to give priority to the higher-listed requirement over requirements of lower priority.

Req 1. `muttest` shall be applied to all existing Rust projects following the recommended project setup [Rus23]. This is critical for practical applications of `muttest` in real-world projects, since most projects have been developed without mutation testing in mind. In particular, editing large parts of the codebase to satisfy the restrictions imposed by `muttest` is infeasible for large projects. This requirement entails that `muttest` supports all features of the Rust programming language. For seamless integration into existing workflows, `muttest` provides a `cargo` plugin so that programmers can use the command `cargo muttest` to perform mutation analysis. Moreover, the implementation of `muttest`

does not rely on unstable language features, ensuring best-possible compatibility with future versions of Rust.

Req 2. `muttest` shall not generate invalid mutations. Evaluating mutants of invalid mutations skews the mutation score for strong and weak mutation analysis, incurs avoidable effort for the programmer when analyzing the mutation analysis report, and negatively impacts the performance of mutation analysis by running more test cases than necessary. Invalid mutations can occur when applying a context sensitive mutation operator to a code fragment that does not meet the relevant requirements. For Rust programs, this includes the following typical cases.

- Mutation of a literal into a value outside the range allowed by its type
- Mutation of an operator into an operator that is not supported by the operands' types
- Removal of expressions and statements that are critical to the correctness of the control- or data flow of a function

Req 3. `muttest` shall incorporate the following techniques to reduce the cost of mutant evaluation because prior research found that the cost of mutation testing is the main cause for restrained industry adoption [UM10].

- Mutations are baked via source code transformations to reduce the number of times the compiler is called.
- Data from coverage analysis and the results of weak mutation analysis are used to determine which test is relevant for the evaluation of which mutant.
- Users can choose the level of detail in which mutation analysis is conducted.
- Users can choose the timeout duration after which a mutant is considered killed.

Req 4. `muttest` shall support as many mutation operators as possible while satisfying the requirements above. Additionally, `muttest` provides a helpful mutation analysis report based on the chosen level of detail.

Req 5. The project `muttest` shall itself be written in a modular and maintainable manner. Additionally, the functionality of `muttest` is tested using appropriate testing methods.

2.2 The Key Design Decisions for Implementing `muttest`

In the following, we explain the parts of the design and architecture of `muttest` that are critical for meeting all these requirements.

To ensure best-possible integration with existing projects, we use core features of the Rust programming language throughout `muttest`. Rust's stability-guarantees promise

```rust
// original
x + y

// transformed
match get_mutation(1) {
    "+" => x + y,
    "-" => x - y,
    "*" => x * y,
    // ... other mutations
}
```

Listing 2.1: An example of mutation baking via source code transformation inspired by Untch, Offutt, and Harrold [UOH93]. This listing shows a Rust function and a possible transformation.

forward-compatibility for these features for future versions of Rust. Specifically, we implement the source code transformation with a procedural macro that is activated by annotating the code under test with the attribute `#[mutate]`. We use the Rust compiler itself as a static program analysis engine and utilize the gathered information to determine which mutations are valid and should be evaluated. Further, we apply the technique of autoderef-based specialization to implement context-sensitive mutation operators without introducing compiler errors.

The method of baking mutations by transforming source code is inspired by Untch, Offutt, and Harrold [UOH93]. Listing 2.1 illustrates this idea, showing an example of a source code transformation. A run-time check that controls the behavior of the code fragment is introduced, making it possible to enable one of the baked mutations dynamically. This technique allows the evaluation of all mutants generated by this process after compiling the test suite and the code under test once. In order to satisfy requirement 1, the source code transformation must be able to correctly process any valid Rust source code. In particular, the transformation must be designed in such a way that it never introduces compiler errors and supports all features of the Rust programming language.

The source code transformation inserts instrumentation for dynamic program analysis alongside the baked mutations. The gathered data is used for determining which mutations are covered and which mutants survive weak mutation analysis. We combine this strategy with an additional procedural macro, enabled by the attribute `#[muttest::tests]`, that transforms each test case to get fine-grained control over its execution. This allows `muttest` to track which test covers which mutation and to selectively skip the execution of any test case that is not relevant for the mutant under evaluation. Consequently, mutants that are not covered by any test case require no further steps to determine that they survive both strong and weak mutation analysis.

Users of `muttest` can configure the level of detail for each run of `cargo muttest`. This configuration enables `muttest` to complete the command more quickly by skipping

analysis steps that are not of interest. The following three levels are supported.

1. Perform weak mutation analysis only.
2. Evaluate each mutant using strong mutation analysis.
3. Compute the full kill matrix, i.e. compute which test case kills which mutant.

Finally, we structure the code of the `muttest` project in a way that each mutation operator can be defined in a single file, independently from other mutation operators. To ensure the composability of all mutation operators, they have to follow the guidelines described in Section 3.1. The correctness of the implementation of each mutation operator is ensured by unit tests in which their functionality is tested in an isolated manner. We apply `muttest` to itself to improve the quality of its own test suite.

2.2.1 The Automated Process to Perform Mutation Analysis

The design decisions outlined above determine the process for mutation analysis by generating and evaluating mutants. The command `cargo muttest` performs the following three steps.

1. The test suite and the code under test are compiled. During compilation, the compiler transforms all parts of the code that are annotated with `#[mutate]`, baking mutations based on detected syntax patterns. The tests annotated with `#[muttest::tests]` are instrumented to allow fine-grained control over their execution and to enable tracking of run-time behavior of the code under test for each test case separately. The information about generated mutations and the marked tests is stored as a compilation artifact alongside the executable test suite. The next steps use these results without calling the compiler again.
2. The test suite is run without any active mutations to check that all tests pass and to record the behavior of the mutable code and their run-time. During this first run of the test suite, the static program analysis information generated by the compiler is extracted and is used to determine which mutations are valid. The data from dynamic program analysis is used to determine which mutations each test case covers and kills for weak mutation analysis. A preliminary mutation analysis report is generated, presenting the results of weak mutation analysis and its mutation score as well as enumerating which mutations are covered.
3. If requested by the user, all mutants are evaluated whether they survive or are killed in strong mutation analysis by running tests while activating the corresponding mutation. Only tests that cover the mutated code fragment and for which the mutant did not survive weak mutation analysis are executed in this step. Unless a kill matrix is requested, the evaluation of each mutant stops once a single test fails. Finally, a mutation analysis report is compiled, containing the strong and weak mutation scores and the evaluation of each mutant.

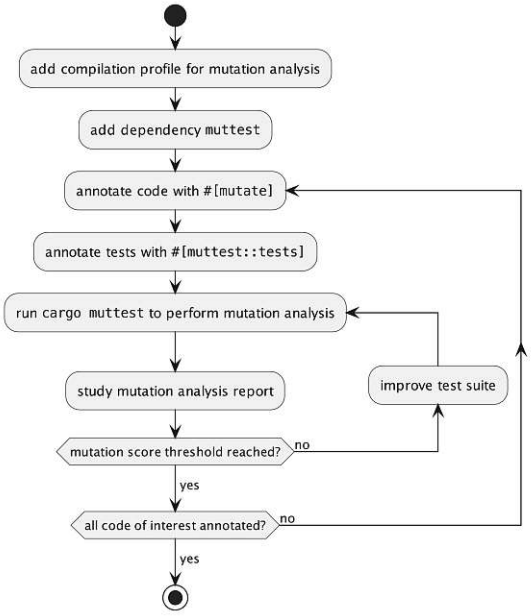

Figure 2.2: The iterative process of mutation testing.

2.2.2 The Mutation Testing Process for Programmers

To perform mutation testing effectively, programmers should apply `muttest` in an iterative process outlined in Figure 2.2. Informally, the results of mutation analysis are use to incrementally improve the test suite until a desired mutation score is reached.

For large projects, it is beneficial to enable compiler optimizations when compiling the code under test and the test suite because they are only compiled once but executed multiple times. To achieve this, a new compilation profile is introduced that includes the properties `opt-level=3` and `lto=true`. All subsequent calls to `cargo muttest` then need to enable this profile.

Programmers enable mutation analysis for a given crate by adding the dependency `muttest`. Then, the annotation `#[mutate]` is used to choose which regions of source code to include for mutation analysis. The modules containing relevant tests have to be annotated with `#[muttest::tests]`. This opt-in approach allows targeted and tailored mutation analysis and helps to focus on parts of the project for which test suite quality is more important. The command `cargo muttest` automatically generates and evaluates mutants and compiles a mutation analysis report.

In an iterative process, the programmer reads the mutation analysis report, improves the test suite, and re-runs the mutation analysis process. By studying the compiled report, the programmer can analyze which parts of the code suffer from test suite deficits. To kill previously survived mutants, the code has to be manually analyzed whether the mutant is equivalent and what inputs cause a test case to fail, if possible. The test

suite can be improved either by writing new tests, adapting test data, or strengthening assertions in existing test cases. Choosing the level of detail of the mutation analysis report appropriately can help to find more easily detectable test suite weaknesses more quickly. This partially automated process is repeated until satisfactory mutation scores for strong and weak mutation analysis are reached.

Mutation analysis can be applied in the continuous integration process of a software project. For example, the command `cargo muttest --threshold 90 --weak-threshold 95` fails if either the strong mutation score is below 90% or the weak mutation score is below 95%. Using `muttest` this way, minimal values for strong and weak mutation scores can be enforced.

Mutation testing users should not include `muttest` in release artifacts outside the testing environment. There are two common approaches to avoid this.

1. Declaring `muttest` as a `dev-dependency` allows mutation testing for unit tests. Instead of using `#[mutate]` and `#[muttest::tests]` directly, programmers annotate their code with `#[cfg_attr(test, muttest::mutate)]` and `#[cfg_attr(test, muttest::tests)]`, respectively.
2. To perform mutation testing for integration tests as well as for unit tests, `muttest` is declared as an optional dependency. Programmers should then use the attributes `#[cfg_attr(feature = "muttest", muttest::mutate)]` and `#[cfg_attr(feature = "muttest", muttest::tests)]`. It is important that the `muttest` feature flag is not activated outside testing environments.

We continue to refer to the attributes as `#[mutate]` and `#[muttest::tests]` and imply that programmers use one of these two methods.

2.3 The Architecture of `muttest`

Figure 2.3 shows an overview of the crates of the `muttest` system and their dependencies. The architecture is greatly influenced by the process described above and the features and restrictions provided by the Rust programming language. The commitment to the mechanism of procedural macros has a particularly significant impact.

Most of the functionality of `muttest` is implemented in the crate `muttest-core`. It contains the definition and implementation of mutations and transformation rules to detect and bake the supported mutations. Additionally, it provides functionality for activating mutations and for performing dynamic program analysis by recording the behavior of relevant parts of the target code.

Ideally, we want the `muttest` system to be a single crate that provides all functionality. However, the Rust programming language requires that a procedural macro is declared in a crate that cannot contain code that is executed at run-time. This forces the use of a

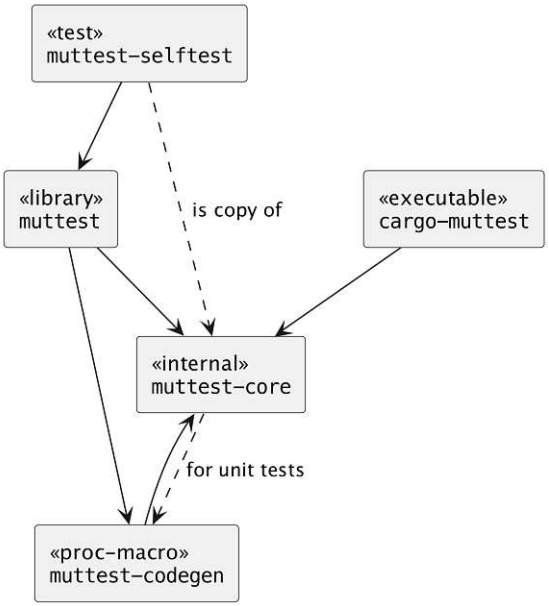

Figure 2.3: The crates of the muttest project and their dependencies.

separate crate we named `muttest-codegen`. This crate uses the transformations implemented in `muttest-core` to define the procedural macro. To hide these implementation details, we reexport all relevant definitions from the core library as well as the procedural macros in the `muttest` crate, which users are supposed to add as the only dependency in the projects they want to perform mutation testing in.

The core crate contains infrastructure to define and execute unit tests that can check the behavior of mutations in an isolated manner. For exhaustive and meaningful tests, the correctness of the transformation rules is tested in combination with the functionality for mutation activation and program analysis. This requires that the test suite of `muttest-core` uses the procedural macro defined in `muttest-codegen`, introducing the circular dependency shown in Figure 2.3. The build system for Rust projects handles this situation by building the crate `muttest-core` without tests and uses it for processing `muttest-codegen`. Then, when building the test suite for `muttest-core`, the procedural macro is available. This process of compiling the crate `muttest-core` twice only occurs when running its test suite. It is not performed when programmers use `muttest` in their projects.

The crate `muttest-selftest` is a copy of `muttest-core` and is used to perform mutation analysis on the code and test suite of the core library itself. It depends on `muttest` to mirror the setup that typical projects need when running mutation analysis. For correct results, it is necessary that mutants of `muttest-selftest` are evaluated using the original code of `muttest-core`. We are aware that an error in `muttest-core` could cause the incorrect evaluation of mutants as killed, hiding the error as well as corresponding

flaws in the test suite. However, we find this additional layer of quality assurance helpful despite this possible weakness.

Prior work [Bog18] has found that some crates introduce compiler errors after baking mutations. If a crate provides implementations for certain traits that impact the results of the type inference algorithm, adding it as a dependency can introduce compiler errors to previously well-typed code. In particular, this situation occurs when using the popular crate `serde_json` for serializing and deserializing data in JSON format, which defines implementations of `PartialEq` that allow the comparison of their internal `Number` type with builtin types. Specifically, the expression `&[0u8][..] != []` fails to compile when importing this crate but is otherwise accepted by the compiler. As a result, `muttest-core` depends on as few external crates as possible.

2.3.1 Communication between Components

The different components of the `muttest` framework need to exchange information in order to accomplish their tasks as described above. The process `cargo-muttest` uses environment variables to configure invocations of the procedural macro and the executions of the test suite, which respond by saving relevant data to files. The compiler artifacts for Rust projects are stored inside a folder called `target`. We therefore place all information relevant for mutation analysis inside the folder `target/muttest` created by the main process `cargo-muttest`. In order to inform the running test suite and the procedural macro of this folder, `cargo-muttest` sets the environment variable `MUTTEST_DIR` appropriately. If this environment variable is not set, no source code transformation is applied and no mutations are activated.

If the crate for which mutation analysis should be performed has already been compiled without generating mutations, `cargo-muttest` is unable to meaningfully proceed. To avoid this situation, we force the recompilation of the code under test and trigger the execution of all source code transformations by placing a constant with the value `option_env!("MUTTEST_DIR")` inside the `muttest` crate. This causes the compiler to track changes to that environment variable. Importantly, only `muttest` and the code under test are recompiled as a consequence. The crates `muttest-core`, `muttest-codegen`, and all other dependencies of the user's projects are not recompiled.

Each invocation of the procedural macro, triggered by the use of the attribute `#[mutate]`, searches for patterns of code that can be mutated. Detected fragments that admit mutations are reported in files of the form `mutables-<crate>.csv` within the `target/muttest` folder, where `<crate>` is replaced by the name of the crate the attribute appeared in. The distinction between different crates is necessary because a Rust project can contain multiple crates to which mutation analysis is applied and for each crate, mutations are enumerated independently.

In order to activate a mutation for a test suite run, `cargo-muttest` sets the environment variable `MUTTEST_MUTATION`. Additionally, the variable `MUTTEST_TARGET` is used to define in which crate the given mutation should be activated. If no mutation is activated,

the type information generated by the compiler during compilation is written to the file `types.csv` in the folder `target/muttest`, while dynamic program analysis data is written to `coverage.csv`. However, if a mutation is activated, the content of the previously generated file `coverage.csv` is read to determine which tests are relevant for evaluating the activated mutation.

2.3.2 Structure of the Procedural Macro `#[mutate]`

The procedural macro `#[mutate]` bakes mutations by transforming the source code in the following steps.

1. The compiler calls the procedural macro on all code fragments annotated with `#[muttest]`.
2. The source code is parsed into an abstract syntax tree using the `syn` library.
3. The abstract syntax tree is traversed in postorder using `syn::Fold`, searching for mutable patterns of code.
4. Each matching syntax node is transformed according to the mutant schemata approach, baking all mutations by adding run-time checks and inserting instrumentation for dynamic analysis.
5. The information gathered during the transformation is saved as additional compiler artifacts.

2.3.3 Testcase Transformation

Tests that do not cover activated mutations cannot impact the evaluation of a mutant. Skipping these tests improves the efficiency of mutation analysis without loss of accuracy. We achieve this optimization by gathering the appropriate data during the initial run of the test suite with no active mutations. Specifically, we record what mutations are covered and how the mutated code fragments behave for each test case individually. This data is then used for deciding which tests to run when evaluating mutants.

We implement the skipping of tests by transforming each test case using a procedural macro, activated by adding the attribute `#[muttest::tests]`. It inserts a statement that controls the execution of the test at the beginning of each test. The significant parts of this implementation are shown in Listing 2.4. At the beginning of each test case, we create a token that contains all relevant information for the test case and informs the run-time context of `muttest` that the test starts. This token is dropped when the test terminates either by completing successfully or by panicking, which triggers the report of the test end via the token's `Drop` implementation. This way, the `muttest` core library can determine which test is currently running when recording coverage and behavior data.

In line 6 of Listing 2.4, the statement contains an option to skip the test by returning early. This is triggered when the test suite is run while evaluating a mutant for which this test case is not relevant. For tests marked with `#[should_panic]`, the early exit is

```rust
// This statement is inserted at the beginning of each test
// by the `#[muttest::tests]` attribute.
let Some(_test_token) =
        CONTEXT.start_test(
            MuttestTest { /* test-specific information */ })
        else { if should_panic { panic!() } else { return } };
        // the else branch skips the test without failing

// The destructor is called the end of the test.
impl Drop for MuttestTestToken {
    fn drop(&mut self) {
        let success = std::thread::panicking() == self.should_panic;
        CONTEXT.end_test(self, success);
    }
}
```

Listing 2.4: The components used for tracking the start and end of each test, simplified. The construction of `MuttestTest` includes information about the test case, such as id, name, the module the test occurs in, and whether the test is marked with `#[should_panic]`.

implemented via a `panic!()` instead of a `return` statement. If the user did not request the computation of a kill matrix, the test suite is terminated after the first failing test.

By default, tests not annotated with `#[muttest::tests]` are not considered as part of the test suite since they do not allow fine-grained dynamic program analysis. To avoid running these tests, the transformation renames the marked tests to contain the string `__muttest_test_`. We use the feature of Rust test suites to filter test cases based on their name to only execute the desired tests. This requires the programmer to refrain from using this special string as part of test case names. We provide a configuration option to run all test cases including tests without annotation. This mode gives correct results, but suffers a performance penalty because these tests cannot be skipped based on the recorded dynamic behavior.

As tests that take longer than expected are considered killing the mutant under evaluation for strong mutation analysis, accurately detecting their timeouts is important for the correctness and performance of mutation analysis tools. We want to avoid the situation that a mutant is killed via timeout but the test suite would pass within a time frame that does not raise the suspicion of the programmer. In such cases, the test suite should not be considered failing. To achieve this, the programmer can set their desired timeout threshold.

Timeouts for long-running test cases are enforced by a dedicated watchdog thread that regularly checks if a test has reached its timeout. The test suite is terminated once a timeout is detected, killing the mutant under evaluation. During the first run of the test

```rust
#[test]
fn test_cmp() {
    #[mutate_isolated("binop_cmp")]
    fn min(a: i32, b: i32) -> i32 {
        if a < b { a } else { b }
    }

    let data = data_isolated!(min);
    assert_eq!(data.mutables.len(), 1);

    let result = call_isolated! {f(1, 2)};
    assert_eq!(result.res, 1);
    assert_eq!(result.mutable_behavior(1), "LT");

    assert_eq!(call_isolated! {min(1, 2) where 1:">"}.res, 2);
}
```

Listing 2.5: A unit test for the mutation operator for comparison operations.

suite, we verify that no test case violates its timing restriction.

2.4 Quality Assurance of muttest

In order to improve and ensure the quality of muttest, we implement a system for testing the behavior of mutation operators using unit tests. Moreover, we apply muttest to itself to compute the mutation score of its own test suite.

2.4.1 Unit Tests

For all mutation operators, unit tests check that

- the correct code patterns are detected,
- the mutations can be activated and behave as expected, and
- the transformation does not introduce compiler errors.

Listing 2.5 shows a typical unit test. The test case contains a function that is annotated with #[mutate_isolated] to trigger the source code transformation. The attribute specifies which mutation operator is tested. The macro data_isolated! extracts the information regarding detected code patterns. The macro call_isolated! is used to execute the local function while performing dynamic program analysis and to optionally activate a mutation for that execution. Using these macros, each unit test can be defined and run independently from all other unit tests.

```toml
# point to the `lib.rs` file of `muttest-core`
[lib]
path = "../src/lib.rs"

# enable "selftest" feature
[features]
default = ["selftest"]
selftest = ["muttest-codegen/selftest"]
```

Listing 2.6: Excerpt from the crate manifest for `muttest-selftest`

2.4.2 Mutation Analysis of `muttest`

We apply `muttest` to the crate `muttest-core` by introducing the crate `muttest-selftest`. An additional crate is necessary because mutated code cannot correctly activate its own mutations. Listing 2.6 shows relevant parts of the crate manifest `Cargo.toml` of `muttest-selftest`. By setting the `path` configuration option to point to the same location as for `muttest-core`, it is ensured that both crates have the same content.

Mutation testing of `muttest-core` is enabled by the feature `selftest`, which is important so that users of `muttest` are not affected by the transformations `muttest` applied on itself. Consequently, the parts of code included in mutation analysis are annotated with the attributes `#[cfg_attr(feature = "selftest", muttest::mutate)]` and `#[cfg_attr(feature = "selftest", muttest::tests)]`.

CHAPTER 3

Implementation Techniques for Mutation Operators of `muttest`

We describe the core ideas underlying the implementation of mutation operators in the mutation analysis tool `muttest`. First, we give guidelines that implementations of mutation operators must follow in order to satisfy the requirements and fit into the architecture discussed in the previous chapter. We focus on technical challenges introduced by using the strategy of using a procedural macro for baking mutations. Second, we explain how certain features of the Rust programming language can be used to implement mutation operators. Finally, we show how tracking of coverage and weak mutation analysis are implemented.

3.1 Guidelines for Implementing Mutation Operators

The requirements and architecture outlined in the previous chapter induce three main constraints on the implementation of mutation operators for `muttest`.

1. Code generated by the source code transformation must be syntactically correct, well-typed, and satisfy Rust's rules for ownership and borrowing.
2. All valid mutations can be activated using the concept of baked dynamic mutations presented in Section 3.2.
3. If the validity of baked mutations cannot be checked from within the procedural macro, their validity has to be reported at run-time if the mutable is reached by a test case using the mechanism described in Section 3.3.

In order to prevent syntax errors, all transformations are written in a way that the transformed code can replace the original code in-place. In particular, transformations

of expressions have to yield new expressions that are valid in the same position in the program. In most cases, the transformations can be written such that they depend on the section of the transformed code, not on its context in the abstract syntax tree. Notable exceptions are given in Section 3.1.1, which have to be excluded from the mutation baking process.

Type errors are avoided by preserving the types of expressions. This requires careful construction of the transformation rule because the procedural macro itself has no access to the types of the target expressions. The main challenge is to construct a transformation in a way such that the type inference algorithm yields the same results as for the original code. One strategy to achieve this is shown in Listing 3.4, which uses the type variable `I` to force the literal and the transformed `run(..)` expression to have the same type. Generating dead branches are another powerful means that can be used to introduce type-level equations that force the type inference algorithm to reach the desired result [Boga]. This idea can be combined with the use of variables of type `PhantomData` to leverage the compiler for type analysis. Transformations for context sensitive mutation operators can be implemented with autoref-based specialization. This technique allows fallback implementations in the case that the types do not satisfy the properties required by the mutation operator. We illustrate both ideas in the transformation for the mutation of `+=` into `-=` in Listing 3.7.

Mutation operators that target patterns of code built from subexpressions like the arguments `a` and `b` of the calculation `a + b` need to preserve the data flow of the program. In particular, due to Rust's ownership rules, it is generally incorrect to evaluate an expression twice. For example, if a value is consumed within the expression, it cannot be used again. Furthermore, it can be invalid to delete statements or expressions that diverge since diverging branches are allowed to destroy values that would otherwise be used later in the function. For these reasons, source code transformations must be constructed in a way such that all subexpressions are called exactly once.

Each baked mutation must provide a possibility to activate its mutations by using the `get_action` function. This function has the signature `fn(MutableId) -> &'static str`, meaning that all mutations have to be encoded as a string. For mutation operators that support weak mutation analysis, their behavior needs to be recorded to determine which mutations would behave identically. For context sensitive mutation operators, it is also required to extract the information which mutation is valid in case the mutable is covered.

The advantage of transformations that obey these guidelines is that they can be defined independently from each other. By traversing the abstract syntax tree in postorder, child nodes are processed prior to their parents. Since subexpressions are not deeply inspected during pattern matching, earlier transformations applied to children do not get in the way of the later transformations of their parents.

```rust
const X1: u8 = 1; // value `1`
static X2: u8 = 2; // initializer of static variable `2`
let x2 = [0; 4]; // length of array `4`
fn sum(xs: [u8; 4]) -> u8; // const generic `4`
fn one() -> i32 { 1 } // type `i32`
const fn one() -> i32 { 1 } // body of const function
id!(1) // macro argument `1`
```

Listing 3.1: Examples of code in which dynamic activations of mutations are not possible. Code that cannot be mutated by run-time switches is highlighted by the comments.

3.1.1 Limitations of Baked Mutations

The semantics of the Rust programming language imposes several limitations on mutation operators that are implemented with the method of baking mutations. The dynamic activation of mutations make it impossible to mutate code that is evaluated at compile-time, such as constants, types, and arguments of macros. Listing 3.1 shows some of the most important examples. Mutations of these fragments of code might still be valid, but would necessarily require recompilation. The arguments of some macros are also evaluated at compile-time and without detailed knowledge of the macro's implementation, it cannot be determined if run-time activated mutations are possible. This means that code in these contexts must be excluded from source code transformation and, as a consequence, from mutant generation.

The source code for run-time activated baked mutations always have a structure similar to `if is_activated() { mutated } else { original }`. The typing rules for Rust programs require that both branches have the same type. This implies that mutations that change the type of expressions are not possible. For example, the statement `let x = 1u8;` cannot be mutated into `let x = 1i32` without recompiling the program even though this mutation might be valid. Moreover, the mutation of types of function arguments and the type of the return value can not be implemented by source code transformations. This restriction does not typically apply to dynamically typed, interpreted languages, but is common for compiled languages with statically checked types.

3.2 Baked Dynamic Mutations and *Mutables* for Strong Mutation Analysis

Some patterns of source code allow a large set of mutations. This is particularly true for literals of string, character, and numeric values, as changing the value of a literal typically results in a correct program. It is infeasible to evaluate all possible mutants. Existing mutation analysis systems based on mutation baking address this issue by baking a small set of supported mutations from which one can be selected during mutant evaluation [Jus14; Bog18]. An example of such statically enumerated mutations for character literals is shown in the first example of Listing 3.2. In these systems, it is necessary to recompile

```rust
// original code
let c = 'X';

// baked enumerated mutations
let c = match active_mutation_id() {
    1 => 'A',
    2 => '🦄',
    _ => 'X',
};

// baked dynamic mutation
let c = match MutableId(1).get_action() {
    Some(mutation) => to_char(mutation),
    None => 'X',
};
```

Listing 3.2: Baked statically enumerated and dynamic mutations for character literals, simplified. The baked enumerated mutation contains a set of possible mutations that is fixed at compile-time, while for the baked dynamic mutation, the behavior can be defined at run-time.

the program when changing the sets of mutations, which limits the flexibility of the mutation analysis system.

We develop the concept of baked dynamic mutations to address this shortcoming and illustrate our solution in Listing 3.2. We introduce the concept of a *mutable*, which is a fragment of code whose behavior can be altered. In contrast, a mutation is a fragment of code and an alternative behavior, while a mutant is the resulting program when applying the mutation. Instead of selecting from a set of mutations defined at compile time by calling the function `active_mutation_id`, the function `get_action` is used to query the framework's run-time library how this code fragment shall behave. The answer `None` signals that the code fragment shall behave as if unchanged, while the response `Some(mutation)` activates a mutation and contains a description of the desired alternate behavior. In this branch, the variable `mutation` has a value of type `&str`, so we use the partial function `to_char` to perform the appropriate type conversion. Using this method of baked dynamic mutations, `muttest` allows the definition of an alternative behavior for each mutable on every run of the test suite. This is used to evaluate mutants with strong mutation analysis, in which all relevant test cases are run while while its mutation is activated.

In `muttest`, all mutables are assigned numeric identifiers instead of enumerating the mutations, which is common in other mutation analysis systems. Instead, a mutation is described by the id of the original code, the `MutableId`, in combination with an alternate behavior in the form `<MutableId>=<alternate-behavior>`. While the enumerated baked mutation into `'A'` is assigned id `1` in Listing 3.2, the corresponding baked dynamic

```
let x = 255; // x: u8
let y = 255; // y: i32
f(x, y); // f: fn(u8, i32)
```

Listing 3.3: The variables `x` and `y` are defined identically. The function `f` requires the two arguments of type `u8` and `i32` respectively. Consequently, type inference yields different types for the variables and their corresponding integer literals.

mutation is described by `1=A`.

3.3 The Compiler as Static Analysis Engine

Several mutation operators require static analysis to guarantee valid mutations. For example, mutations of integer literals into values outside the range allowed by their type are invalid. The type inference of Rust programs causes that an analysis of syntax patterns alone is not sufficient to determine the types of literals. Listing 3.3 illustrates this situation using the function `f` as well as two variables `x` and `y`. Even though their definitions are lexically identical, they get assigned different types via type inference based on their use within the call of function `f`. The variable `x` and its corresponding literal have type `u8`, an 8-bit unsigned integer, while `y` and its literal are determined to have type `i32`, a 32-bit signed integer. Consequently, the mutations of the second literal into the values `256` or `-1` are valid, while they are not for the first literal.

In general, the procedural macro has no access to the signature of the function `f` and therefore cannot determine the types of the variables `x` and `y`. This example shows that static analysis based on code fragments alone without the considering the context they occur in, is not enough to determine the set of valid mutations. Moreover, all definitions and declarations given in the same compilation unit or any of its dependencies can influence the type inference of a given code fragment.

We develop a strategy to utilize features of the type system of the Rust programming language to determine the set of valid mutations in `muttest`. The compiler already performs all of the required static analyses during compilation. However, the compiler does not answer the specific requests required by mutation analysis. Importantly, it does not output the types of expressions of interest for mutation analysis. Instead, it can only produce executable binary files and libraries from Rust source code. By constructing traits and defining the source code transformation in a deliberate way, we can extract the answers to the relevant queries.

Listing 3.4 shows our approach to type analysis for integer literals. The trait contains the function `type_str` that returns the string-representation of the literal's type. Instead of transforming the literal into a `match` expression, the literal is replaced by a function call to `run` containing the original literal as the second argument. This function has a generic parameter for the type of the original literal and returns a value of the same type. When called, the type of the literal is reported to the mutation analysis run-time using the

```rust
pub trait MutableInt {
    fn type_str() -> &'static str;
    fn parse(m: &str) -> Self;
}
// implementations for all integer types generated by a declarative macro
// here only for `u8`
impl MutableInt for u8 {
    fn type_str() -> &'static str { "u8" }
    fn parse(m: &str) -> Self {
        m.parse().expect("unable to parse mutation")
    }
}

fn run<I: MutableInt>(id: MutableId, original: I) -> I {
    id.write_types(I::type_str(), &[]);
    return match id.get_action() {
        Some(mutated) => I::parse(mutated),
        None => original,
    }
}

// original code
let x = 7;

// transformed
let x = run::<_>(MutableId(1), 7);
```

Listing 3.4: Compiler-based analysis of types for integer literals and their baked dynamic mutation, simplified. The trait `MutableInt` defines two functions: `type_str` for static analysis and `parse` to implement the baked dynamic mutation.

`write_types` method. Then, the function returns either the value of the original literal or a mutated value set by the result of the `get_action` function, analogous to the baked dynamic mutation of character literals shown in Listing 3.2.

Using the compiler this way, it is only possible to learn the type information if the `run` function for the mutable is executed during the first run of the test suite without active mutations. If the type is never reported, then the mutation is not covered by any test case and no mutations are evaluated because they are all either invalid or would survive. This method of static analysis ensures that only valid mutants are evaluated by `muttest` regardless of the behavior of the test suite.

```
1  // original
2  a += b;
3
4  // transformed
5  match MutableId(1).get_action() {
6      None => a += b,
7      Some(_) => a -= b,
8  }
```

Listing 3.5: Naive implementation of the mutation operator mutating `+=` into `-=`. This transformation introduces a compilation error for line 7 in case the `-=` operation is not implemented for the types of `a` and `b`.

3.4 Context Sensitive Mutations for Overloaded Operators

Rust allows overloading of many operators for arbitrary types of their operands. Generally, the mutation of one operator into another is only valid if the other operator is overloaded for the same combination of types. Analogous to types of integer literals described in the section above, the types of variables and expressions depend on the results of the type inference algorithm and cannot be extracted from the program text of the mutable alone. This renders the mutation that alters such operators context sensitive. In the following, we discuss how to implement the mutation of the operator `+=` into `-=`.

A naive transformation that bakes the mutation, changing the operator `+=` into `-=`, is shown in Listing 3.5. It yields a compilation error if the operation `-=` is not supported for the combination of types of `a` and `b`. For example, this mutation is not valid if the variables `a` and `b` are string values, while being valid if they are numerical values.

We implement this mutation operator in a way that does not introduce type errors by using trait `MaySubAssign` presented in Listing 3.6. The trait and its two implementations combine the technique of autoderef-based specialization with the strategy of extracting type information from the compiler. The first implementation is used in cases where the operation `-=` is possible. This is ensured by the bound `L: AddAssign<R>` in line 5. In contrast, the second implementation has no bounds in the corresponding line 15. It serves as a fallback because it implements the trait for the referenced version of the tuple of phantom types. The method `is_sub_assign` can be used to query whether subtraction is possible, while the method `do_sub_assign` performs the operation, if possible. The trait uses `PhantomData` as the target for specialized methods, which is necessary to include both types in the consideration for method selection. Values of type `PhantomData` contain type information but contain no run-time data and therefore have no performance impact.

The source code transformation that bakes this mutation is shown in Listing 3.7. The baked dynamic mutation is implemented in the `else` branch and relies on the trait `MaySubAssign` and its functions. We use the technique of using dead branches introduced by Bogus [Boga] to guide the type inference algorithm to yield the same results as for

```rust
trait MaySubAssign<L, R> {
    fn is_sub_assign(&self) -> bool;
    fn do_sub_assign(&self, l: &mut L, r: R);
}
impl<L: AddAssign<R>, R> MaySubAssign<L, R>
    for &(PhantomData<&mut L>, PhantomData<R>)
{
    fn is_sub_assign(&self) -> bool {
        true
    }
    fn do_sub_assign(&self, l: &mut L, r: R) {
        *l -= r;
    }
}
impl<L, R> MaySubAssign<L, R>
    for (PhantomData<&mut L>, PhantomData<R>)
{
    fn is_sub_assign(&self) -> bool {
        false
    }
    fn do_sub_assign(&self, l: &mut L, r: R) {
        panic!()
    }
}
```

Listing 3.6: Trait for the context sensitive mutation into `-=` and two implementations.

the original program. We augment Bogus' method with the function `with_type`, which gives a value of `PhantomData` of the appropriate type alongside the value. Prior to the execution of the statement, we report the validity of the mutation into `-=`. Then, the expressions on both sides of the `+=` operation are evaluated, after which we declare that the operation is covered by the current test case. It is important that we do not consider the operation `+=` as covered if one of the expressions terminate the execution of the current statement, for example by exiting the function early. Finally the behavior of the mutable is chosen based on the result of `get_action`. The branch representing the mutation to `-=` calls the function `do_sub_assign`, which performs the subtraction, if possible, but does not introduce compilation errors if subtraction is not supported because a fallback implementation is provided. This process for evaluating mutations assures that the mutation is never activated in case a mutation is determined to be invalid.

It is noteworthy that this strategy for specialization performs method resolution based on the types that are declared and inferred and does not depend on the concrete types at run-time. The Rust community plans to introduce specialization inspired by Haskell's type classes, in which the method is resolved for each concrete type separately. In this

```rust
let (mut left_type, mut right_type) = (PhantomData, PhantomData);
if false {
    *with_type(left, left_type) += with_type(right, right_type);
} else {
    MutableId(1).write_types("",
        &["-=", (&&(left_type, right_type)).can_sub_assign()]
    );
    match MutableId(1).get_action() {
        None => *left += right,
        Some(_) => {
            (&&(left_type, right_type)).do_sub_assign(left, right)
        }
    }
}
```

Listing 3.7: Transformation of `left += right`, simplified. The dead first branch of the `if` statement guides the type inference algorithm to yield the same types compared to the original program and allows extraction of the types of the arguments of the `+=` operator into the `left_type` and `right_type` variables.

```rust
fn double<T: AddAssign<T> + Clone>(a: &mut T) {
    *a += a.clone();
}
```

Listing 3.8: Example function that doubles a value by incrementing it by itself. The mutation to change `+=` into `-=` is invalid without the constraint `T: SubAssign<T>` in the method signature. This is true even if this function is only called with types that support `-=`.

proposal, we could define a trait with methods similar to the ones given in Listing 3.6. When applying this implementation to the generic function in Listing 3.8, the transformation reports incorrect results. The function `double` can be called with various types, some of which may support subtraction beside addition. In these cases, the method `is_sub_assign` would then return using this form of specialization. However, the mutation of `+=` into `-=` is not valid because the type `T` is not bound by `SubAssign<T>` and the mutant would fail to compile. Therefore, we find autoref-based specialization to be the ideal means for baking context sensitive mutations and discard specialization based on concrete types as a possible implementation strategy for this use case.

3.5 Coverage Tracking and Weak Mutation Analysis

Tests that do not cover a code fragment cannot have an impact on the survival of any of the corresponding mutants in strong or weak mutation analysis. In order to determine these, we record which test case covers which mutable by invoking the method

```rust
fn run<L: PartialOrd<R>, R>(
    m_id: MutableId,
    left: &L,
    right: &R,
) -> bool {
    let cmp = left.partial_cmp(&right);
    m_id.write_behavior(cmp);
    match m_id.get_action() {
        None => cmp.is_lt(),
        Some(_) => cmp.is_le(),
    }
}

// original
a < b

// transformed
run::<_, _>(MutableId(1), &a, &b)
```

Listing 3.9: Implementation of the baked mutation for `a < b` into `a <= b`, simplified. Instead of calling one of the comparison operators, `<` or `<=`, `run` calls the function `partial_cmp` from the trait `PartialOrd`.

`write_coverage` right before calling `get_action`. During the first run of the test suite without any mutations active, the coverage data is collected during the execution of each test case. This data is subsequently used to determine which tests should be included in the evaluation of which mutant during strong mutation analysis. This method call has been omitted from the listings discussed in this chapter so far.

The mutation analysis tool `muttest` supports weak mutation analysis for certain mutation operators by recording the behavior of the mutable during the first run of the test suite. We illustrate weak mutation analysis based on the mutation of the expression `a < b` into `a <= b`. Its implementation is given in Listing 3.9. Instead of evaluating the original expression `a < b` directly, the ordering of the values `a` and `b` is determined using the method `partial_cmp`. The function call `write_behavior` then reports the result of the comparison. All such outcomes that occur during the execution of a test case are gathered and collected as the set of behaviors of the original code fragment. If this set contains the result that equal values are compared, then the mutation of the operator `<` into `<=` is regarded as killed in weak mutation analysis since they would evaluate to different results for these inputs.

Not all mutation operators support meaningful weak mutation analysis. For example, mutations of literals always give a different value than the original program. Mutants generated by these mutation operators are always regarded as killed in weak mutation analysis, if covered.

CHAPTER 4

Comparative Evaluation of Rust Mutation Analysis Tools

We evaluate `muttest` in comparison to `mutagen` and `cargo-mutants`, two other mutation analysis tools for Rust. We investigate the quality and run-time of the mutation analysis performed by each of these systems. To this end, we first adapt and apply the framework for comparing the features of mutation analysis tools developed by Amalfitano et al. [Ama+22]. We then perform mutation testing with each tool for a small example program and examine the resulting mutation analysis reports as well as the test cases constructed during the mutation testing process. Further, we apply each tool to selected well-known projects of the Rust ecosystem and investigate the performance and quality of mutation analysis. Finally, we discuss the findings of this comparative evaluation.

We evaluate the mutation analysis systems using the version `1.74.0-nightly` of Rust released on 25$^{\text{th}}$ August, 2023. The performance benchmarks are run on a Linux virtual machine with 4 cores of an Intel Xeon E5-2620 2GHz processor and 16GB main memory. Unfortunately, the tool `mutest-rs` did not compile successfully and we could not find sufficient documentation regarding its design. We had to exclude it from this evaluation.

4.1 Overview of Features

To get an overview of differences and similarities of the features of the mutation analysis tools `muttest`, `mutagen`, and `cargo-mutants`, we adapt the framework for comparing mutation analysis tools for Java programs introduced by Amalfitano et al. [Ama+22]. The section regarding implemented mutation operators is expanded, while criteria not relevant to Rust programs are omitted. The results are summarized in Tables 4.1 and 4.2.

Table 4.1: Comparison of Rust mutation analysis tools regarding version, deployment, and mutation process

	`muttest`	`mutagen`	`cargo-mutants`
Version			
Release Version	0.1	0.2	23.6.0
Release Year	2023	2022	2023
License	MIT	Apache 2.0 or MIT	MIT
Depoloyment			
Rust version	≥1.72	nightly	≥1.65
Unstable features	–	specialization	–
Build integration	`cargo muttest`	`cargo mutagen`	`cargo mutants`
Multi crate support	✓	–	✓
Mutation Process			
Mutation mechanism	Baking	Baking	Source file edit
Target code selection	attribute `#[mutate]`	attribute `#[mutate]`	CLI arguments
Test selection	attribute `#[muttest::tests]`	all tests	all tests
Mutant inspection	by LOC	by LOC	by LOC
Kill matrix	optional	–	–
Performance Improvements			
Mutation baking	✓	✓	–
Skip non-covered mutants	✓	✓	–
Run tests	only covering & weakly-killing tests	all	all
Weak mutation analysis	✓	–	–
Parallel execution	–	–	✓
Timeout	static threshold	5x slower, min 500ms	5x slower, min 20s
Mutant Selection	–	–	–
Mutant Ranking	–	–	–
Equivalent mutant prevention	–	–	–

Table 4.2: Comparison of Rust mutation analysis tools regarding user-centric features and mutation operators

	`muttest`	`mutagen`	`cargo-mutants`
User-centric features			
User interface	CLI	CLI	CLI
Input	Rust workspace	Rust workspace	Rust workspace
Text-based log	✓	✓	✓
Report	single file	single file	multiple files
Report format	JSON	JSON	plain text & JSON
Documentation quality	sufficient	sufficient	good
Mutation operators			
Literals	✓	✓	–
Match arm guards	✓	–	–
Comparison operators			
Swap `==` ↔ `!=`	✓	✓	–
Swap `<` ↔ `<=` ↔ `>` ↔ `>=`	✓	✓	–
Arbitrary swap of comparison operators	✓	–	–
Arithmetic operators			
Swap `+` ↔ `-`	✓	✓	–
Swap `*` ↔ `/`	✓	✓	–
Swap shifts `>>` ↔ `<<`	✓	✓	–
Arbitrary swap of binary arithmetic operators	✓	–	–
Unary operator deletion	✓	✓	–
Statement Deletion			
Method call	–	∽	–
Arbitrary Statements	–	–	–
Extreme mutations			
Return `Default`	✓	–	✓
Panic	✓	–	–
Extended	–	–	✓

All three tools have been released quite recently. `muttest` and `cargo-mutants` were released in 2023, while the last release of `mutagen` was in 2022. The latest release of `mutagen` is version 0.2. `cargo-mutants` is released using date-based versioning `CalVer`[1]. The numbers contained in its latest version `23.6.0` refer to the year and month of the release, respectively. The system `muttest` as developed as part of this thesis is released as version 0.1. This version number was chosen because we expect to adapt the user interface of `muttest` based on feedback from the Rust community. All three tools are released under open-source licenses. `muttest` and `cargo-mutants` are licensed under the MIT[2] license, while `mutagen` is dual-licensed under MIT and Apache License Version 2.0[3].

The systems `muttest` and `cargo-mutants` do not depend on unstable language features and can be used on stable Rust, while `mutagen` requires the unstable feature `specialization` and therefore requires a beta version of the Rust compiler. All three tools offer a cargo plugin for performing mutation analysis. `muttest` and `cargo-mutants` support mutation analysis of multiple crates within a single workspace, while `mutagen` can only perform mutation analysis on a single crate.

`muttest` and `mutagen` use the mechanism of baking to implement mutations and use attributes for selecting which parts of code should be included in mutation analysis. `muttest` additionally requires the programmer to annotate the tests. `cargo-mutants` edits the source code files directly and performs mutation analysis without the need for such attributes. Of the compared systems, only `muttest` supports the generation of a kill matrix.

`muttest` and `mutagen` both use mutation baking for improving run-time of mutation analysis and skip the evaluation of mutants that are not covered by any test. `muttest` also skips the execution of test cases that do not cover the mutant under evaluation or let it survive weak mutation analysis. `cargo-mutants` is the only system that evaluates mutants in parallel to improve mutation analysis performance. All tools detect timeouts for mutants that are not expected to terminate. `muttest` uses a fixed threshold for the run-time of test cases. `mutagen` and `cargo-mutants` apply an adaptive strategy for test cases that assumes that a test case does not terminate if it runs five times longer than evaluating a mutant. Both systems have a minimum timeout, which is 20 seconds for `cargo-mutants` while `mutagen` uses 500ms. No system ranks mutants by priority, allows to evaluate a selected subset of generated mutants, or employs mechanisms for detecting and preventing equivalent mutants.

All systems provide a command-line interface to perform mutation analysis and require a Rust workspace as input. Further, they write a log of the mutation analysis process to the terminal and save a machine readable form of the analysis report in JSON format. The tool `cargo-mutants` also provides plain text files to summarize which mutants are

[1] `https://calver.org`
[2] `https://opensource.org/license/mit/`
[3] `https://opensource.org/license/apache-2-0/`

killed and which have survived. The documentation quality of `muttest` and `mutagen` are sufficient and the documentation of `cargo-mutants` is good[4].

`mutagen` implements mutations on literals, a set of operator-swaps for arithmetic calculations, and the deletion of unary operators. `cargo-mutants` only implements extreme mutations, replacing the body of a function to return a default value. `cargo-mutants` generates additional mutations for functions returning a known set of types, replacing the implementation of functions by returning one of several possible values. `muttest` implements mutations of literals, arbitrary swaps of binary arithmetic operators, and the deletion of unary operators. It also supports extreme mutations that replace the body of a function with a default value or cause the function to always panic. Another novel mutation operator alters the guards of match arms to always block or always allow the execution of the guarded arm. `mutagen` also supports deletion of statements that consist of a single method call, but this mutation operator can cause a compiler error in some situations [Bogb]. No system supports mutations that delete arbitrary statements.

4.2 Mutation Testing on a Small Example

DeMillo, Lipton, and Sayward [DLS78] introduce several small programs for analyzing mutation analysis systems and these examples have since been part of demonstrations of numerous such systems [JH11; Ama+22]. We use one of these programs for this evaluation and port the Fortran code to Rust in a way that closely preserves the original program to allow future research to compare the results meaningfully with results from other languages.

Listing 4.3 shows a function that classifies triangles based on the length of their sides. The program validates the input parameters, performs calculation and comparisons, and returns the classification of the triangle as a string. Although this function's logic is simple, it contains 23 mutable code fragments supported by the mutation analysis tools under consideration.

- 4 equality operators
- 4 comparison operators
- 5 arithmetic operators (2 additions, 3 multiplications)
- 2 boolean operators
- 7 string literals
- the function body for extreme mutation

We apply all three mutation analysis tools according to their documentation. Table 4.4 shows the mutations generated by each mutation analysis tool. `muttest` generates 110 mutants while `mutagen` generates 55. `cargo-mutants` only generates two mutations because it only supports extreme mutations. Both replace the entire body of the function by a constant string. Of all generated mutants, four are equivalent.

[4] `https://mutants.rs`

```rust
fn triangle(x: u32, y: u32, z: u32) -> &'static str {
    if x > y || y > z {
        return "lengths not sorted";
    }
    if x + y <= z {
        return "illegal";
    }
    if x == y || y == z {
        return if x == z { "equilateral" } else { "isosceles" };
    }
    let x2y2 = x * x + y * y;
    let z2 = z * z;
    if x2y2 == z2 {
        return "right angled";
    }
    if x2y2 < z2 {
        return "obtuse angled";
    }
    return "acute angled";
}
```

Listing 4.3: A procedure to classify triangles, ported to Rust from Fortran from Ramamoorthy, Ho, and Chen [RHC76]. The attributes are feature gated.

Table 4.4: Mutations generated for each mutation operator by each of the mutation analysis tools

	`muttest`	`mutagen`	`cargo-mutants`
Equality operator	20	4	–
Comparison operator	20	12	–
Arithmetic operators	45	5	–
Boolean Operator	2	2	–
String literal	21	21	–
Extreme Mutation	2	–	2
Total Mutations	110	44	2
Equivalent mutants	4	1	0
Non-equivalent mutants	107	43	2

Table 4.5: Minimal set of test cases that kill all non-equivalent mutants generated by `muttest`

Test Case	x	y	z	classification
T_1	1	2	3	illegal
T_2	3	3	7	illegal
T_3	4	4	3	lengths not sorted
T_4	3	4	5	right angled
T_5	3	4	6	acute angled
T_6	2	4	5	obtuse angled
T_7	3	4	4	isosceles
T_8	3	3	4	isosceles
T_9	3	3	3	equilateral

- The three checks for equality in lines 8 and 9 are never executed with the left side greater than the right side, because of the check in line 2. Therefore, the mutations of the operators `==` into `>=` produce equivalent mutants. These mutants are only generated by `muttest`
- The comparison operator in line 18 contains a comparison in which equal values are never compared because of the check in line 13. Therefore, the mutation of the operator `<` into `<=` gives an equivalent mutant. This mutant is generated by `muttest` and `mutagen`

Table 4.5 lists the test cases necessary to kill all 106 non-equivalent mutants generated by `muttest`. To kill all 43 non-equivalent mutants generated by `mutagen`, the test cases T_2 and T_8 can be omitted. Both mutations generated by `cargo-mutants` are killed by any of the test cases.

4.3 Benchmark on Real-world Projects

We apply the three tools `muttest`, `mutagen`, and `cargo-mutants` to a selected set of well-known projects of the Rust ecosystem and evaluate the applicability of these systems to real-world software projects. We select projects based on popularity on the library index lib.rs[5] as well as their size and choose the following projects.

- `clap`: command-line argument parsing[6]
- `tokio`: asynchronous computation run-time[7]
- `rand`: random number generators[8]

[5] `https://lib.rs/`
[6] `https://github.com/clap-rs/clap/`
[7] `https://github.com/tokio-rs/tokio/`
[8] `https://github.com/rust-random/rand/`

- `regex`: regular expression engine[9]

Table 4.6: Overview of key characteristics of the real-world projects included in this benchmark.

project	version	crates	files	lines of code	# of tests	test suite build time	test suite run-time
clap	4.3.24	9	318	51903	1377	8m 20s	0.7s
tokio	1.32.0	13	675	76643	1243	1m 10s	0.5s
rand	0.8.5	6	89	14091	269	14s	0.2s
regex	1.9.4	8	216	93725	364	2m 4s	0.2s

Table 4.6 outlines the scale of each project and their test suite. We perform mutation analysis on each of these projects in these steps:

1. Download the latest release of the projects. We use the latest release published on the github repository.
2. Remove long-running tests meant for performance- or stress testing.
3. Fix errors regarding type inference that are introduced by the source code transformations of `muttest` and `mutagen`.
   - Expressions containing shifts, like `(1 << 2u32) - 3u16` cause type inference errors that are fixed by annotating the type of the first literal.
   - We adapt `mutagen` so that it ignores constant methods.
4. Compile and run the test suite.
5. Apply each of the mutation analysis tools according to their documentation.
   - We set a timeout of 5 seconds per test case for mutation analysis with `muttest`.
   - We select the largest crate of the project for `mutagen`, since this tool does not support mutation analysis of multiple crates at once.
   - We enable the parallelization feature `cargo-mutants`, allowing four concurrent jobs.
6. Generate and evaluate all mutants. We record and measure the following aspects of the mutation analysis process:
   - number of generated mutants
   - number of invalid, survived, and killed mutants
   - mutation score
   - run-time to build the test suite
   - run-time of strong and weak mutation analysis, excluding the build-time of the test suite
   - average evaluation-time of mutants

[9] `https://github.com/rust-lang/regex/`

We repeat the experiment four times and compute the averages of the measured run-times. The results of the benchmark are collected in Table 4.7.

4.4 Discussion

The qualitative comparison of the three mutation analysis tools for Rust programs, `muttest`, `mutagen`, and `cargo-mutants` shows that `muttest` is more powerful than the other systems. In particular, `muttest` supports the most mutation operators, without relying on unstable features of the Rust programming language. It also incorporates the most performance improvement strategies.

By conducting mutation testing on the function `triangle`, we have shown that additional mutations can be helpful for designing a test suite of high quality. While `muttest` requires all nine test cases of Table 4.5 to reach the optimal mutation score, `mutagen` and `cargo-mutants` only require seven and one, respectively. We claim that all of these nine test cases are valuable for detecting possible programming errors that could be introduced by implementing the `triangle` function. Therefore, omitting any of these test cases reduces the quality of the test suite. Specifically, a single test case is not fit to check the correctness of the implementation of this function. We have also seen that among the additional mutations generated by `muttest`, some of them were equivalent, which raises the cost of mutation testing due to the manual effort required to detect these mutants as unkillable.

The results of the benchmark show that `muttest` generates the most mutations for all four projects. We expect that additional mutations help to improve the test suite quality by the mutation testing process. Moreover, `muttest` and `mutagen` both evaluate mutations significantly quicker than `cargo-mutants`. This difference is mainly caused by the technique of baking mutations. We believe that the performance gap between `muttest` and `mutagen` is mainly caused by their different handling of timeouts.

It is notable that all three tools compute different mutation scores. Detailed analysis of the projects and their test suites is necessary to determine which of these values measures the real test suite quality most accurately. In particular, the number of equivalent mutants is unknown as this number cannot be determined automatically. Moreover, `muttest` does not generate context sensitive mutations if the static analysis information could not be extracted because their mutables were not covered by the test suite, which further skews the mutation score.

Table 4.7: Results of benchmark of mutation analysis tools of real-world projects.

	`muttest`	`mutagen`	`cargo-mutants`
`clap`			
Mutations	2487	973	1815
- invalid	0	N/A	393
- killed	715	184	242
- survived	1772	789	1177
Score	28.7%	18.9%	17%
Run-time			
- build test suite	7m 14s	9m 13s	–
- weak mutation analysis	1.7s	–	–
- strong mutation analysis	21s	14s	3h 26m
- per mutant	8.5 ms	14ms	6s
`tokio`			
Mutations	4201	1744	472
- invalid	0	N/A	371
- killed	1799	253	0
- survived	2402	1491	101
Score	42.82%	14.5%	0%
Run-time			
- build test suite	4m 5s	1m 31s	–
- weak mutation analysis	1.3s	–	–
- strong mutation analysis	12m 57s	5m 4s	33min 67s
- per mutant	184ms	174ms	4.3s
`rand`			
Mutations	2937	200	490
- invalid	0	N/A	117
- killed	1747	100	139
- survived	1190	100	234
Score	59.5%	50%	35.4%
Run-time			
- build test suite	1m 54s	36s	–
- weak mutation analysis	4s	–	–
- strong mutation analysis	9m 47s	16s	6m 58s
- per mutant	186ms	83ms	853ms
`regex`			
Mutations	9566	2698	4642
- invalid	0	N/A	1212
- killed	4490	785	1406
- survived	5076	1913	2024
Score	46.9%	29.1%	41%
Run-time			
- build test suite	5m 0s	3m 4s	–
- weak mutation analysis	9.8s	–	–
- strong mutation analysis	20m 11s	5m 34s	11h 24m
- per mutant	126ms	124ms	8.9s

CHAPTER 5

Conclusion

The aim of this thesis was to develop a tool for mutation analysis of Rust programs that improves on the state of the art. Leveraging static and dynamic program analysis was critical for achieving this objective. Our qualitative and experimental comparison of `muttest` with related state-of-the-art tools `mutagen` and `cargo-mutants` for Rust show that we achieved our overall goal.

We designed our approach carefully to ensure its sustainability. Omitting dependencies to unstable features of Rust ensures best-possible compatibility with future versions of Rust. Our implementation is instead based on core language features alone. Specifically, we use procedural macros for source code transformations, leverage the compiler as a static analysis engine, apply autoderef-based specialization, and guide the type inference algorithm towards the desired result with the insertion of dead branches. As a result, the mutation operators do not generate invalid mutations and `muttest` is compatible with all Rust projects that can be built and tested with the standard build system, `cargo`. Using this architecture, we implemented most mutation operators that are implemented in `mutagen` and `cargo-mutants` and introduced several new ones. Following the guidelines described in Section 3.1, it is possible to add further mutation operators to `muttest`.

We integrated a novel optimization into `muttest` that allows faster evaluation of mutants in strong mutation analysis by introducing fine-grained dynamic program analysis. In contrast to related systems, `muttest` does not only skip test cases that do not cover the activated mutation, but also skips test cases which let the mutant survive weak mutation analysis.

Conceptually, we extended the map of mutation testing by introducing the distinction between code that can be mutated, mutables, and mutations. This allows the implementation of baked dynamic mutations and to group properties of mutations targeting the same code, such as coverage and results of program analysis. Moreover, this architec-

ture increases the modularity of the implementation of `muttest`. We had not seen this abstraction in previous systems.

The project `muttest` consists of 5706 lines of code and contains 107 tests. When performing mutation analysis of itself, 211 mutations were generated, of which 167 were killed. This results in a mutation score of 79.15%. We released the source code of the project on github[1].

5.1 Future Work

The project `muttest` can be extended in various directions in future research.

Usability. The mutation analysis report generated by `muttest` is currently text-based. To be applicable in practice, its results should be presented in a visual interface instead. Other systems like `pitest` [Col+16] provide a web-based view of the mutation analysis report. We expect that industry adoption is significantly hindered by the lack of such an interface.

Streamlining mutation testing with `muttest`. For developing `muttest`, we focussed on defining as many mutation operators as possible. Future research can investigate which of these are the most helpful in practice and which are sufficient for accurately measuring test suite quality. Such studies have been conducted for tools designed for other programming languages [NAM08; Off+96] and their findings cannot be applied directly to mutation testing for Rust programs. The results of an empirical study on sufficient mutation operators for Rust could reduce the cost of mutation testing in practice while keeping its effectiveness for constructing good test cases and evaluating their quality.

Other techniques for mutant evaluation. We explored the impact of program analysis on mutation analysis based on mutation baking. Future studies could investigate other methods for mutant evaluation for mutants of Rust programs. In particular, it should to be possible to adapt the mutation analysis tool `mull`, which is based on LLVM, for Rust. Moreover, it seems worthwhile to look at leveraging `miri`, the interpreter for mid-level intermediate representation (MIR) of Rust programs, to activate mutations at run-time without the need of baking them.

Other approaches for type analysis. The mutation operators in `muttest` rely on generating the appropriate code to extract the type analysis performed by the compiler. Different methods of performing type analysis might allow the implementation of additional mutation operators. In particular, the similarly named project `mutest-rs` could be updated to a newer version of the compiler and compared to `muttest`. Another promising alternative to perform type analysis is to query the compiler frontend `rust-analyzer`. However, we expect that the method of evaluating mutants developed in this thesis is the most likely to be compatible with future versions of Rust.

[1] `https://github.com/samuelpilz/muttest-rs`

Higher-order mutants. When applying multiple mutations at once, the resulting program is a higher-order mutant [JH09]. These composite mutations can represent more complex faults and many test cases that kill first-order mutants are not sufficient to catch certain higher-order mutants. The tool `muttest` can be easily extended to evaluate such mutants by activating multiple baked mutations. However, since the number of higher-order mutations is significantly larger than the number of first-order mutations, generating and evaluating all of them is infeasible in practice. Future work should strive for an algorithm for determining the subset of most helpful ones for mutation testing.

Equivalent mutant detection. The detection of equivalent or redundant mutations is undecidable in general. However, previous research has shown that software verification methods can be used to detect some subset of equivalent or redundant mutations [Had18]. Like all verification methods, these depend heavily on the programming language's semantics, but we conjecture that approaches developed for C can be adapted for Rust.

Mutation analysis and test case generation. In the year 2000, Offutt and Untch [OU01] proposed to combine mutation analysis with automated test case generation methods such as property-based tests and fuzzing. In such a system, the tests are filtered based on the results of mutation analysis. This way, a test suite of high quality can be synthesized in a fully automated manner. Papadakis et al. [Pap+19] define the "Modern Mutation Testing Process", in which the test cases can either be generated automatically or designed manually. However, to the best of our knowledge, this combination is not yet implemented in a mutation analysis tool. We expect that this hybrid approach can greatly improve the confidence of the quality of the automatically generated test cases. We believe it is feasible to integrate a property-based testing library into `muttest` to implement the vision of Offutt and Untch [OU01].

List of Figures

List of Tables

Bibliography

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. "Is mutation an appropriate tool for testing experiments?" In: *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. Ed. by Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh. ACM, 2005, pp. 402–411.

[Acr+79] Allen Acree et al. *Mutation Analysis*. Tech. rep. Georgia Institute of Technology, Sept. 1979.

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[Ama+22] Domenico Amalfitano et al. "How Do Java Mutation Tools Differ?" In: *Commun. ACM* 65.12 (Nov. 2022), pp. 74–89. ISSN: 0001-0782. DOI: 10.1145/3526099. URL: https://doi.org/10.1145/3526099.

[AO17] Paul Ammann and Jeff Offutt. *Introduction to software testing*. eng. Second edition. Cambridge New York, NY Melbourne New Delhi Singapore: Cambridge University Press, 2017. ISBN: 1107172012.

[Boga] Andre Bogus. *A Shifty Riddle*. URL: https://llogiq.github.io/2018/04/11/shift.html.

[Bogb] Andre Bogus. *Arraigning a Statement, vol. 2*. URL: https://llogiq.github.io/2019/03/14/stmt2.html.

[Bog18] Andre Bogus. *mutagen*. 2018–2019. URL: https://github.com/llogiq/mutagen.

[Bud+78] Timothy A. Budd et al. "The design of a prototype mutation system for program testing". In: *American Federation of Information Processing Societies: 1978 National Computer Conference, June 5-8, 1978, Anaheim, CA, USA*. Ed. by Sakti P. Ghosh and Leonard Y. Liu. Vol. 47. AFIPS Conference Proceedings. AFIPS Press, 1978, pp. 623–629.

[Col+16] Henry Coles et al. "Pit: a practical mutation testing tool for java". In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 449–452.

[DH14] Anna Derezinska and Konrad Halas. “Experimental evaluation of mutation testing approaches to python programs”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2014, pp. 156–164.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”. In: *Computer* 11.4 (1978), pp. 34–41.

[DP18] Alex Denisov and Stanislav Pankevich. “Mull It Over: Mutation Testing Based on LLVM”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2018, pp. 25–31.

[Had18] Thomas Hader. “Automated Analysis of Program Mutations using Bounded Model Checking”. Bachelor’s Thesis. TU Wien, 2018.

[How82] William E. Howden. “Weak Mutation Testing and Completeness of Test Sets”. In: *IEEE Transactions on Software Engineering* 8.4 (1982), pp. 371–379.

[JH09] Yue Jia and Mark Harman. “Higher Order Mutation Testing”. In: *Information and Software Technology* 51.10 (2009). Source Code Analysis and Manipulation, SCAM 2008, pp. 1379–1393. ISSN: 0950-5849.

[JH11] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Trans. Software Eng.* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62. URL: https://doi.org/10.1109/TSE.2010.62.

[JP17] Jacques-Henri Jourdan and François Pottier. “A Simple, Possibly Correct LR Parser for C11”. In: *ACM Trans. Program. Lang. Syst.* 39.4 (2017), 14:1–14:36. DOI: 10.1145/3064848. URL: https://doi.org/10.1145/3064848.

[Jus14] René Just. “The Major mutation framework: Efficient and scalable mutation analysis for Java”. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, July 2014, pp. 433–436.

[Kal] Lukas Kalbertodt. *Generalized Autoref-Based Specialization*. URL: http://lukaskalbertodt.github.io/2019/12/05/generalized-autoref-based-specialization.html#using-autoderef-for--two-specialization-levels.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2019.

[KO91] K. N. King and A. Jefferson Offutt. “A Fortran Language System for Mutation-based Software Testing”. In: *Software: Practice and Experience* 21.7 (1991), pp. 685–718.

[Le+14] Duc Le et al. “MuCheck: an extensible tool for mutation testing of haskell programs”. In: *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. Ed. by Corina S. Pasareanu and Darko Marinov. ACM, 2014, pp. 429–432.

[Lév22] Zalán Bálint Lévai. *mutest-rs*. 2022. URL: `https://github.com/zalanlevai/mutest-rs`.

[MMP13] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. “Efficient JavaScript mutation testing”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 74–83.

[NAM08] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. “Sufficient mutation operators for measuring test effectiveness”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 351–360.

[NJW16] Rainer Niedermayr, Elmar Jürgens, and Stefan Wagner. “Will my tests tell me if I break this code?” In: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery, CSED@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 23–29.

[Off+96] A. Jefferson Offutt et al. “An Experimental Determination of Sufficient Mutant Operators”. In: *ACM Trans. Softw. Eng. Methodol.* 5.2 (1996), pp. 99–118.

[OU01] A. Jefferson Offutt and Roland H. Untch. “Mutation 2000: Uniting the Orthogonal”. In: *Mutation Testing for the New Century*. Ed. by W. Eric Wong. Boston, MA: Springer US, 2001, pp. 34–44. ISBN: 978-1-4757-5939-6.

[Pap+19] Mike Papadakis et al. “Chapter Six - Mutation Testing Advances: An Analysis and Survey”. In: ed. by Atif M. Memon. Vol. 112. Advances in Computers. Elsevier, 2019, pp. 275–378.

[RHC76] Chitoor V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. “On the Automated Generation of Program Test Data”. In: *IEEE Trans. Software Eng.* 2.4 (1976), pp. 293–300. DOI: `10.1109/TSE.1976.233835`.

[Rus23] Rust Project. *Cargo*. 2015–2023. URL: `https://doc.rust-lang.org/cargo`.

[sou23] sourcefrog. *cargo-mutants*. 2023. URL: `https://github.com/sourcefrog/cargo-mutants`.

[Tol] David Tolnay. *Autoref-based stable specialization*. URL: `https://github.com/dtolnay/case-studies/blob/master/autoref-specialization/README.md`.

[UM10] Macario Polo Usaola and Pedro Reales Mateo. “Mutation Testing Cost Reduction Techniques: A Survey”. In: *IEEE Softw.* 27.3 (2010), pp. 80–86. DOI: `10.1109/MS.2010.79`.

[UOH93] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. “Mutation Analysis Using Mutant Schemata”. In: *Proceedings of the 1993 International Symposium on Software Testing and Analysis, ISSTA 1993, Cambridge, MA, USA, June 28-30, 1993*. Ed. by Thomas J. Ostrand and Elaine J. Weyuker. ACM, 1993, pp. 139–148.

[Woo93] M.R. Woodward. “Mutation testing—its origin and evolution”. In: *Information and Software Technology* 35.3 (1993), pp. 163–169. ISSN: 0950-5849.