Master's Thesis in Computer Science

Search-based Test Suite Generation for Rust

Vsevolod Tymofyeyev

May 31, 2022

Examiners:
Prof. Dr. Gordon Fraser
(Chair of Software Engineering II)

Prof. Dr. Christian Hammer
(Chair of Software Engineering I)

Vsevolod Tymofyeyev:
Search-based Test Suite Generation for Rust
Master's Thesis, University of Passau, 2022.

Abstract

Rust is a robust programming language which promises security and performance at the same time. Despite its young age, it has already convinced many and has been one of the most popular programming languages among developers since its first release. However, mistakes are a fact of life, and like any other software, Rust programs need to be tested extensively.

In this work, we propose the first search-based tool called RustyUnit for automatic generation of unit tests for Rust programs. It incorporates a compiler wrapper, which statically analyzes and instruments a given program to generate and evaluate tests targeting high code coverage. To achieve it, the tool employs a genetic algorithm which iteratively evolves a test suite by recombing and mutating test cases.

Furthermore, we provide an empirical study on the performance of our approach using 6 real-world open-source Rust libraries and compare the results to a baseline random search algorithm.

Contents

1 Introduction

In the programming language world, there are two major sides: low-level languages, which offer better performance at the expense of security and high-level languages, which provide security for programmers through certain constructs such as garbage collection that lead to runtime overhead. The young programming language Rust tries to combine the good from both. This statically typed language for system programming promises a similarly high performance as C++ while maintaining extended type and memory safety by default. It ensures invariants at compile-time, which means that abstractions (so-called zero-cost abstractions) and automatic memory management are not associated with runtime costs as is the case with managed languages. Rust prevents (among others) the following common problems: dangling pointers, data races, integer overflow, buffer overflows, and iterator invalidation. Only the integer and buffer overflows are checked at runtime, whereas the buffer overflows can be reduced to static checks using iterators [2]. This symbiosis makes the language particularly attractive to developers, as it has seen a rise in popularity rankings for several years in a row [81]. Even the most prominent corporations consider using Rust in parts of their software. According to Microsoft and Google, 70% of the bugs found in their software in recent years were due to memory leaks caused by widely used insecure languages such as C and C++ [66, 82]. Microsoft, SpaceX, Google, Amazon AWS, and many other companies have already started experimenting with Rust in their products for increased security [9, 13, 79, 82].

1.1. Motivation

Nevertheless, even the Rust compiler cannot guarantee complete correctness, which means that when using this language, one still has to check the quality and behavior of their software, e.g., with unit tests. Unit tests execute minor parts of a program in isolation. Software testing, however, requires specific input data. In most cases, it is the task of a tester or a developer to write tests manually. This procedure is usually very time-consuming and cost-intensive. Sufficiently complex software can have thousands of execution paths driven by different input data. A human can easily overlook some of them.

Moreover, software requirements can change over time and force existing test suites to be manually modified or, in worst case, rewritten from scratch as a result. Thus, covering all possible execution paths is almost impossible in terms of finances and human work [67]. It is assumed that about half of the budget in software projects is spent on testing [12]. Still, developers are often pressed for time (e.g., deadlines on projects) and do not have enough time to test the increasingly complex software despite sophisticated testing tools. This is a big issue because even if some minor bugs only lead to the dissatisfaction of an end-user of a product, some others can cause significant and even health damage [67]. For this reason, many approaches have emerged in recent years and decades to automate this process by generating tests from a given code base [64].

Testing all possible combinations of input data on a program is primarily impossible, though, even in an automated way. It would, in most cases, lead to many equivalent tests which do the same procedure repeatedly. To avoid the effort, one tries to find representative input data for the respective equivalence classes to execute paths determined by certain classes of input data as few times as possible. To this end, many tools exploit symbolic execution. Thereby the System Under Test (SUT) is analyzed and executed symbolically. A set of constraints is defined for the input data necessary to achieve a particular goal in the executed code [24]. Practically, however, symbolic execution means many complex algebraic manipulations, especially when object-oriented containers are involved [55]. Search-based Software Testing (SBST) is an alternative and very active research field whose main idea is to use metaheuristic search techniques to generate a limited number of tests which satisfy a test criterion, e.g., code coverage, within an acceptable time limit [64].

Since Rust is considered young as a stable programming language and appeared in version 1.0 in 2015 [84], there are relatively few options for automatic test generation at the time of writing. These are limited to tools that use symbolic execution to search the possible paths in a given program [17] and random testing tools. That tools exploit the Intermediate Representation (IR) of LLVM, which the Rust compiler produces intermediately during the compilation. However, there is no known use of SBST for Rust as of this writing. SBST is a combination of automatic test generation and metaheuristic search techniques. This subcategory of Search-based Software Engineering (SBSE) resorts to optimization algorithms to solve an NP-hard test generation problem as efficiently and effectively as possible [53]. SBST optimizes a solution concerning a particular objective, which could be, e.g., test case prioritization, test suite minimization, or maximization of real-time properties of the SUT [53]. Search-based techniques for testing have proved to yield good results when applied to programs written in other languages, especially Java. In general, a vast majority of such tools have

been applied to managed languages since the ideas rely on the ability to use some sort of reflection and instrumentation. Java, for instance, is a mature language that provides both. With its reflection capabilities, one can load and inspect the SUT's structure at runtime and run generated tests, which access the SUT on-the-fly within the same process. Java source code is compiled into bytecode, which is a much-simplified representation; there is good tooling support for its instrumentation.

In contrast to Java and other managed languages, which are run within virtual machines, Rust's source code is compiled directly into machine-executable code. It hardly provides any reflection capabilities, so both analysis and instrumentation of the SUT need to be done during the compilation phase, i.e., statically. Another point in which Rust differs strongly not only from managed languages is its affine type system [2], which sets strict rules to how variables can be used. That is, they cannot be accessed and passed around freely. This fact changes the view on how to design a typical program in Rust. The peculiar restrictions of Rust combined with the idea of "chaotic mutations" of tests in SBST pose new challenges to test generation for this language.

1.2. Contributions

With our work, we provide the following contributions:

- We adopt and redefine the problem of automatically generating test cases for Rust programs using a Genetic Algorithm (GA). It can evolve an initial set of randomly generated test cases based on execution feedback and produce a final test suite with high code coverage. The generated tests consider the peculiarities of Rust's type and generic system. Even though the test cases are randomly generated and modified during the process, the output is still valid Rust code.
- We implement our approach as a wrapper around the original Rust compiler and hook into different compilation phases to execute our logic, such as extracting points of interest or instrumenting the SUT. Rust's in-house build system Cargo allows us to provide a custom Rust compiler wrapper when building a SUT. The implementation artifacts and the usage guide are available in a GitHub [1] repository.
- We evaluate our GA-based approach and compare it to a traditional baseline algorithm for test generation, i.e., random search, to obtain certainty about the correctness of the implementation and any performance gain of our approach over the minimum. The evaluation is conducted using a case study built on our own library and a set of 6 real-world Rust libraries.

[1] `https://github.com/foxycom/rusty-unit`

All data, as well as the case study, can be found on our GitHub repository.

1.3. Structure of the Thesis

First, we give an overview of the relevant background on approaches and algorithms that automate tasks in test generation (Chapter 2), which is necessary to understand the concept we build our ideas on. We shall then provide an overview of related work (Chapter 3) regarding test generation with the common approaches. Then, we introduce the basic concepts of the Rust programming language (Chapter 4) and what are its peculiarities in contrast to common programming languages. With the language concepts being presented to the reader, we introduce our tool called RustyUnit for automatic test case generation (Chapter 5) and provide its implementation details and decisions (Chapter 6). To show the performance of our approach, we provide an extensive evaluation (Chapter 7), followed by a summary and an outlook on possible future research directions (Chapter 8).

2

Background

We provide an overview of the concepts to automatic test generation and different categories of techniques that already exist.

2.1. Test Generation in General

Test generation is an actively researched field in science. Ideally, a suite of unit tests that cover all defined coverage criteria can be generated for a program while checking its correctness by implementing automatic oracles for the execution of each unit. An oracle is a mechanism to check whether the output is correct given an input, for example, using a formal specification [63]. Unfortunately, this is often not possible. For one thing, a program may contain execution paths which cannot be achieved under any circumstances. For another, software very rarely has a formal specification that can be used to generate test cases. Thus, in most cases, a developer or tester needs to manually add oracles to the generated test suite (of course, they must know what the correct behavior is) [42]. However, the generated test suite must be kept as small as possible to prevent bloating and be easily comprehensible for humans.

Automatic test generation approaches often employ some coverage criteria as guidelines [36]. A coverage criterion is a collection of test objectives that are typically worked through or covered one at a time, with the necessary input data determined randomly, symbolically, or search-based, for instance. Common coverage criteria are branch coverage, mutation coverage, or error coverage. Branches can be arms of an if expression or a loop header. They are executed when a given boolean expression evaluates to `true` or `false`. Branches can also be nested, which makes a program structure more complex and harder to find suitable values to reach all branches using automatic approaches. In the following, we describe the most common existing methods to automatic test generation split into categories.

2.2. Random Search

Random search is a baseline strategy that relies on replacement of existing solutions. The idea is to repeatedly sample new solutions from the

search space at random, replacing a previous solution with a new one if the new solution is better concerning some predefined coverage criterion. Random search can exploit an archive by employing a specific sampling strategy [21]. *Random testing* is a variant of random search, which builds test suites incrementally. With random testing, the program is executed with random inputs, and the executed structures of the program are observed. Individual test cases are sampled from the search space, and if a test case increases the overall coverage of the test suite, it is kept and otherwise discarded [21].

Since the landscape of fitness values in generating unit tests is reasonably flat and this is a relatively simple search problem, a random search can be at least as effective as evolutionary algorithms and sometimes even better [80].

2.3. Symbolic Execution

Symbolic execution is not an execution of a program in a direct sense. Instead, it assigns symbolic expressions to program variables while tracing a path statically in the program structure [64]. Ideally, symbolic execution of such a path p yields a logical formula ϕ_p, a *path-condition*, describing a set of input data I for the SUT necessary to execute the program P and follow the path p. An Automated Theorem Prover (ATP) determines whether a given path is feasible [24, 54]. If the formula ϕ_p is unsatisfiable, I is empty, and the path p is not feasible. Otherwise, the formula is satisfiable, which results in the set I being nonempty, and the path ϕ_p being feasible [10]. In such case, a model of ϕ_p can provide an example $i \in I$ that can be used in a test to cover the path p. Symbolic execution is a common approach to generating input data or entire unit tests. Many of the tools follow the same principle: instead of running programs with manual or generated input data, the data is populated with symbolic values that can be "anything" initially [17]. Concrete operations on data are replaced by those that can manipulate the symbolic ones. When executing program branches, the tools keep track of the execution of all branches simultaneously. A collection of constraints that must apply to execute each path is stored. If execution ends in a path or the program crashes, a test can be generated from it by using concrete values as input data that satisfy the corresponding path constraints. If the program remains deterministic and unchanged, an execution with concrete input data leads to the same bug in the program.

Dynamic Symbolic Execution (DSE) is an extension of path-based symbolic execution that allows combining concrete and symbolic values to overcome many issues of the original technique [42]. Because of the combination of concrete and symbolic values, DSE is also called concolic execution. The approach starts by executing a program P on some input i, seeding the symbolic execution process with a feasible

path [48, 57]. Then, DSE uses concrete values from the execution $P(i)$ in place of symbolic expressions whenever symbolic reasoning is impossible or undesired [18]. Examples include non-linear arithmetic and cryptographic hash functions that are virtually impossible to solve for symbolic execution [10]. Several tools rely on DSE for automatic generation of tests, for example, KLEE [17], CUTE and JCUTE [78], DART [45], and KLOVER [60].

2.4. Evolutionary algorithms

Even for a simple program, the potential search space for possible input data can be infinite. Evolutionary Algorithms (EAs), which are metaheuristic approaches, promise a cure. These are not closed algorithms per se but strategies that can be adapted to specific problems. For example, a problem-specific fitness function can be defined for test case data generation to compare the quality of possible solutions, i.e., test cases, to the problem [64]. However, metaheuristic search is not only used for test data generation. Other use cases include:

- coverage of the program structure as part of a white-box testing strategy,
- evaluation of a specific program feature according to its formal specification,
- attempts to automatically invoke error conditions or breaks of assertions in a program,
- verification of non-functional features, for example, finding the worst-case execution time of a real-time system.

Evolutionary algorithms are inspired by natural evolution and have been successfully used to address many kinds of optimization problems. In this context, a solution is encoded genetically as an individual; a set of individuals is called a population. The population is gradually optimized by performing nature-inspired operations on individuals, such as mutation, which independently changes components of an individual with a low probability, or crossover, which merges at least two individuals to produce offspring [22]. Genetic algorithms are the best-known instance of evolutionary algorithms; a search using genetic algorithms is performed based on the recombination of intermediate solutions, a mechanism to exchange information between solutions and thus breed new ones [64].

2.5. Genetic Algorithms

Genetic algorithms are the most common form of evolutionary algorithms because they are easy to implement and yield good results on

Algorithm 1 A high level description of a standard genetic algorithm [21]

Input

C	Stopping condition
δ	Fitness function
p_s	Population size
s_f	Selection function
c_f	Crossover function
c_p	Crossover probability
m_f	Mutation function
m_p	Mutation probability

Output

P	Population of optimized individuals

$P \leftarrow \text{GenerateRandomPopulation}(p_s)$
$\text{PerformFitnessEvaluation}(\delta, P)$
while $\neg C$ **do**
  $N_P \leftarrow \{\}$
  while $|N_P| < p_s$ **do**
    $p_1, p_2 \leftarrow \text{Selection}(s_f, P)$
    $o_1, o_2 \leftarrow \text{Crossover}(c_s, c_p, p_1, p_2)$
    $\text{Mutation}(m_f, m_p, o_1)$
    $\text{Mutation}(m_f, m_p, o_2)$
    $N_P \leftarrow N_P \cup \{o_1, o_2\}$
  end while
  $P \leftarrow N_P$
  $\text{PerformFitnessEvaluation}(\delta, P)$
end while
return P

average. The name "genetic algorithm" comes from the analogy between encoding a candidate solution as a sequence of simple components and the genetic structure. This analogy is continued by calling individual solutions individuals or chromosomes [21]. Accordingly, components of a solution are called genes, with possible values of an element called alleles, and their positions in the sequence called locus. Furthermore, an actual encoded representation of a solution manipulated by a genetic algorithm is called a genotype and a decoded: phenotype [64]. Algorithm 1 abstractly shows the mechanism of a standard genetic algorithm. The initial population of size p_s of candidate solutions is typically generated randomly or from a particular seed. Afterward, a pair of individuals are selected from the population using some selection algorithm s_f, such as rank-based, elitism, or tournament selection. Next, both selected individuals are recombined using a crossover operator c_f, e.g., single-point or multiple-point crossover, with a probability c_p to produce two new offspring o_1 and o_2. Then, a mutation operator m_f is applied to both offspring, independently changing the genes with a probability of m_p, which usually is equal to $\frac{1}{n}$, where n is the number of genes in a chromosome. The two mutated offspring are then inserted into the next population. At the end of each iteration, the fitness value of all individuals is computed.

There are many variations of standard GA. For example, in the monotonic variation of standard GA, after mutation and evaluation of the fitness of the offspring, either the best descendants or the best parent chromosomes are added to the next population. In standard GA, both parents and descendants of the following population are added at this point. Another variant of the standard algorithm is steady-state GA, which, like the monotonic variant, retains only the best individuals after a mutation. Instead of creating a new offspring population, it replaces the "parents" with offspring in the original population [21].

The essential points in Algorithm 1 are measuring the fitness of a single chromosome and the search operators. These factors help to evolve a current population into a new one that has more chances to cover a coverage criterion. An individual's fitness determines its ability to survive and participate in the next population, possibly in a mutated form. On the other hand, mutation and crossover operators determine how new individuals are generated from given ones [86]. A *fitness function* controls the selection of individuals so that individuals with good fitness are more likely to survive and participate in breeding offspring. In the literature about search-based test generation, fitness functions are commonly defined as some code coverage criteria, e.g., statement or branch coverage. In the recent years, there has been a trend to optimizing searches for multiple coverage criteria simultaneously. Since coverage criteria typically do not represent conflicting goals, they can be combined in a weighted linear function [76]. However, a high number of coverage goals might negatively affect the performance of an

genetic algorithm. To avoid this, generated tests that cover some targets can be stored in an archive [75], which dynamically adapts the fitness function to guide the search only towards uncovered targets. Generic algorithms can also employ the archive for better effectiveness of search operators. For example, an algorithm could generate further tests by mutating the population from an archive rather than an completely random one [21].

2.5.1. *Crossover*

An essential part of genetic algorithms is the so-called recombination or crossover. This operator loosely models the exchange of genetic information during reproduction in the natural world. It takes two or more parent solutions as input and combines them in a certain way to produce two or more offspring solutions. There are many different types of recombination but the simplest one is one-point recombination [64]. It involves choosing a single point in two solution sequences which separates them into heads and tails, which are then exchanged between the sequences. One-point recombination of two individuals $[0, 255, 0]$ and $[255, 0, 255]$, which would be encoded in binary as 000000011111100000000 and 111111110000111111, respectively, at position 12 would result in the following two offspring [64]:

$$\begin{array}{l|l} 000000001111 & 111100000000 \\ 111111110000 & 000011111111 \end{array} \longrightarrow \begin{array}{l} 000000001111000011111111 \\ 111111110000111100000000 \end{array}$$

2.5.2. *Selection*

A GA can use different selection mechanisms to decide which individuals it should use for a breed. The focus is on the fitness of the individuals. A fitness value can be used directly or scaled a certain way first.

Genetic algorithms store a whole population of solutions instead of only one current solution. The diversity primarily helps to initially sample more of the search space by defining multiple starting points. The population is iteratively recombined and mutated to breed other populations through evolution, known as generations. The idea of recombination is to prefer fitter individuals in the hope of breeding fitter offspring. However, too focused selection on the fittest individuals can lead to them dominating the following generations and thus, limiting the search to only a particular area in the search space due to low diversity. On the other hand, too weak selection may result in a search far too broad to explore, thus, failing to make significant progress toward an optimal solution [64].

Holland's original selection algorithm [51] relied on *fitness-proportional selection*. This mechanism assigned each individual how often a GA

would select it for reproduction based on the fitness of that individual relative to the rest of the population. The process is analogous to a roulette wheel: each individual is assigned a section on the wheel according to its fitness value. The disk is then spun N times to select N parents. At the end of each turn, the position of the marker on the disk determines the individual chosen to be the parent for the next generation. However, fitness-proportional selection can lead to difficulties maintaining constant *selection pressure*. Selection pressure is the probability of the best individual being selected compared to the average likelihood of all individuals being selected. In the first generations of a search, the variance of fitness values of a population is typically high, which leads to the high selection pressure of a fitness-proportional selection. It can lead to convergence much too early. In later generations, when the fitness values of individuals do not differ as much, the selection pressure drops, which might lead to stagnation in the search. Once the parents have been selected, recombination is applied to breed the next generation. Crossover is applied to the selected individuals with probability c_p. c_p is also referred to as crossover rate or crossover probability. The new offspring will be added to the new population if the crossover has been actually applied. Otherwise, the selected parents are copied unchanged into the recent population. After this step, the mutation phase is initiated, which is responsible for reintroducing new genetic material that shall increase the diversity of the search.

Another selection method is the *linear ranking* of individuals, which can overcome the selection pressure issue. This technique sorts individuals according to their fitness, with the best individuals being selected according to their rank [88]. In addition, there is also *tournament selection*, a noisy but fast method to determine offspring. The respective population does not have to be sorted by fitness. Two individuals are randomly selected out of the population. Then, a random real number $0 < r < 1$ is drawn. If r is less than p (where p is the probability that the fitter individual is selected), the individual with better fitness wins and is chosen to be a parent for the following generation; otherwise, the other individual is. Finally, the two individuals are placed back into the population and can be selected again in the next round. The procedure is repeated N times until the required number of parents is selected for the next generation.

2.5.3. Mutation

Mutation involves random modification of offspring to introduce diversity into the search. Traditionally, on binary representation, this consists of flipping bits of each individual at a probability of m_p, where m_p is typically $\frac{1}{n}$, where n, in turn, is the length of the chromosomal bitstring [50]. Even though binary representation is trivially mutable and recombinable, search operators ignore the semantics of solutions,

which means that mutations due to bitflips can lead to invalid solutions with a high probability, depending on the problem. To mitigate this issue, solutions are often processed in their natural state.

2.5.4. *Advanced Individual Encoding*

The simplicity of binary representation of solutions has its price. That form often violates the rule that a solution's neighbor should be reachable with a simple mutation, which is not always true for those encoded with zeros and ones. For instance, the binary encoding of integers 7 and 8 is 0111 and 1000, respectively. To reach the neighboring solution of 7, i.e., 8, the search operator in place would have to change all four bits. Random mutation of binary sequences can also lead to many invalid solutions depending on the search problem. Even though the binary representation of solutions makes the corresponding sequences consist of the minor components (bits) that lead to the most effective results of recombination and mutation, Davis [26] showed in their experiments with genetic algorithms that true solution representations perform better than binary ones. However, the use of true solution representations raises the question of how to implement mutation operator, while recombination only needs the representation of the sequence of components of a solution and can be applied similarly as to a binary solution. On the other hand, a mutation operator must be tailored to the specific problem in any case, e.g., an integer can be trivially replaced by a random one. A more advanced mutation could modify an integer by adding or subtracting some amount. This way, the principle of local search keeps maintained, too [26].

2.5.5. *Objectives*

In general, search-based techniques often resort to such coverage criteria as branch coverage, statement coverage, or mutation coverage. Those are quantitative indications of how good a test suite is. A test suite consists of individual tests, where a test is nothing more than a sequence of variable-length statements. In the Whole Suite (WS) approach, the chosen fitness function measures how close a solution (i.e., a test suite) is to satisfying all coverage criteria or targets. It defines the fitness value as the overall coverage (e.g., branch coverage), that is, the sum of individual distances to coverage targets (e.g., branches) in a SUT. Formally, the problem of finding a test suite that satisfies all coverage targets can be defined as follows [71]: Let $U = \{u_1, ..., u_k\}$ be the set of structural test targets of the program under test. Find a test suite $T = \{t_1, ..., t_n\}$ that minimizes the fitness function:

$$minf_U(T) = \sum_{u \in U} d(u, T), \tag{2.1}$$

where $d(u, T)$ denotes the minimal distance for the target u according to a specific distance function. The goal is to minimize d. $d(u, T) = 0$ holds if the generated test suite T covers the target u. The differences between coverage criteria affect the distance function d, which expresses how far the program execution is from the coverage targets in U when all test cases in T are executed [71].

The targets to be covered for *branch coverage* are conditional branches in a SUT. The optimization goal is to find a test suite that executes all of them, where the function d_b is the traditional branch distance [69] for each branch b to be executed. The problem is formulated as follows [34]: Let $B = \{b_1, \dots, b_k\}$ be the set of branches in a class/module. Find a test suite $T = \{t_1, \dots, t_n\}$ that covers all the feasible branches, i.e., one that minimizes the following function:

$$minf_B(T) = |M| - |M_T| + \sum_{b \in B} d(b, T), \tag{2.2}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods by all test cases in T, and $d(b, T)$ denotes the minimal normalized branch distance for branch $b \in B$. The term $|M| - |M_T|$ refers to the entry edges of the methods not executed by T. The minimum normalized branch distance $d(b, t)$ is defined as follows [42]:

$$d(b,t) = \begin{cases} 0 & b \text{ has been covered} \\ \frac{D_{min}(t \in T, b)}{D_{min}(t \in T, b)+1} & b \text{ has been executed at least twice} \\ 1 & \text{otherwise} \end{cases} \tag{2.3}$$

where $D_{min}(t \in T, b)$ is the minimal non-normalized branch distance, computed according to any of the available branch distance schemes [64]. Minimality refers here to the fact that the condition of a branch can be executed multiple times by the same test or by numerous tests. The minimal distance is favored across all executions [71].

Statement coverage is used to find the optimal solution that executes all statements in a SUT. It is sufficient to execute the branch from which a statement s is directly control-dependent to execute the statement. The distance function d for a statement s can be measured using the branch distance required to execute the branch b and reach s. Formally, the statement coverage function is defined as follows [42]:

$$minf_S(T) = |M| - |M_T| + \sum_{b \in B_S} d(b, T), \tag{2.4}$$

where $|M|$ is the total number of methods, $|M_T|$ is the number of executed methods by all test cases in T, B_S is the set of branches that hold a direct control dependency on the statements in S, and $d(b, T)$ denotes the minimal normalized branch distance for branch $b \in B_S$ [71].

Strong mutation coverage is also common in the literature. Its coverage targets are mutants, variants of the original program, artificially

created by injecting modifications that mimic actual bugs. In this case, the optimal test suite is the one that can cover (or kill) all mutants in a SUT. A test case *strongly kills* a mutant only in the case when the observable state of an object or a return value of a method differs between the original and mutated programs. This condition is typically checked using automatically generated assertions. Generated tests that have high mutation coverage are beneficial for detecting regressions.

2.5.6. *Many-objective Search*

In their most basic version, evolutionary algorithms, such as the $1 + (\lambda, \lambda)$ [30] or $\mu + \lambda$ [83] ones, perform a single-objective search; that is, they try to optimize for one goal at a time and typically yield only one solution (or multiple solutions with the same optimal fitness value [71]). For instance, an algorithm might choose a random objective and attempt to generate a test covering the selected objective. A maximum time limit, or *search budget*, is often set to keep the search within a specific time frame. The one goal at a time approach necessarily divides the budget uniformly among all coverage goals. However, this is a problem because some goals will need more or less budget than others, depending on how big the effort is to find a corresponding test. Some goals may even be unreachable, e.g., due to conflicting constraints, which wastes the entire budget searching for a test. Fraser's and Arcuri's [42] WS approach tries to overcome this challenge by optimizing entire test suites rather than individual test cases. The idea is to merge all the individual fitness values of test cases in a test suite into a single aggregated fitness value using scalarization [28]. Thus, a problem involving multiple objectives is transformed into a traditional single-objective and scalar one by summing up each target's corresponding minimum fitness values.

This way, all coverage targets are considered simultaneously during the search. The sum of all fitness values of test cases is the overall fitness value of a test suite. The goal is to apply single-objective algorithms such as GA to an intrinsically multi-objective problem. Even though the WS approach is more effective than one objective at a time, it still suffers from the well-known problem of sum scalarization in many-objective optimization [28]. On the other hand, many-objective algorithms have been shown to yield better results for some single-objective problems. The main idea is to split a single complex objective into several simpler ones, which lowers search's probability of getting stuck in a local optimum. Nevertheless, two important issues must be considered when applying many-objective optimization to the problem of test generation: (i) no currently available multi- or many-objective solver scales to the number of targets that occurs in coverage testing of real software [6]; (ii) multi-objective solvers are designed to

increase diversity in solutions, not to reach every single objective (i.e., reduce fitness to 0), as is necessary for test generation [71].

MOSA

Test generation is intrinsically a many-objective problem which often defines the goal as coverage of multiple targets (e.g., branches) [71]. Multi-objective algorithms like Non-dominated Sorting Genetic Algorithm II (NSGA-II) [29] have successfully solved software engineering problems with two or three objectives, e.g., software refactoring and test case prioritization. However, these algorithms encounter scaling limitations when solving problems with more than three objectives [59]. Multiple many-objective algorithms have been proposed in the literature to overcome the scalability issue focusing on selection pressure. Panichella et al. [72] introduced the prominent Many-Objective Sorting Algorithm (MOSA), which defines each coverage criterion as an independent optimization objective. According to the authors, previous algorithms were not applied to the many-objective test generation problem because they were not scalable enough for the number of targets in genuine software.

Moreover, the main idea of such algorithms is primarily to find a tradeoff solution in the objective space, while in test generation, only those solutions that cover one or more uncovered targets are of interest [71]. MOSA is a many-objective GA that has been specifically tailored to the problem of test generation. Its three main features are: (i) an innovative preference criterion is used instead of ranking solutions based on the Pareto optimality; (ii) the search is performed only on the yet uncovered coverage targets; (iii) all test cases that satisfy one or more targets not previously covered are stored in an archive, which contains the final test suite at the end of the search.

Rojas et al. [75] have presented the Whole Suite with Archive (WSA) Approach, a hybrid strategy that combines elements of MOSA with the traditional WS Approach. Thus, WSA still incorporates sum scalarization of an entire test suite and exploits an archive of individual test cases. Furthermore, its search is focused only on previously uncovered coverage targets, too. Experiments showed that WSA is statistically more effective than WS and one-objective at a time approaches. However, from the theoretical point of view, test suites are not evolved at all during the process since the final test suite is not the best individual from the last generation of GA but is artificially synthesized from tests in the archive.

DynaMOSA

The main weakness of MOSA is that it considers all objectives equal and independent. Structural programs are fraught with dependencies, though. For example, some goals can only be satisfied if another one

has been covered at that point [42]. To better explain this concept, let us consider the example in Listing 2.1.

Listing 2.1 A nested function with control dependent blocks

```
fn foo(a: i32, b: i32, c: i32) -> i32 {
  let mut x = 0;
  if a == b {   // b1
    if a > c {  // b2
      x = 1;
    } else {    // b3
      x = 2;
    }
  }
  if b == c {   // b4
    x = -1;
  }
  x
}
```

The four branches to be covered, b_1, b_2, b_3, and b_4, are not independent. In fact, coverage of b_2 and b_3 can be achieved only after $b1$ has been covered since the former targets are under control of the latter. In other words, there is a control dependency between b_1 and (b_2, b_3), which means that the execution of (b_2, b_3) depends on the outcome of the condition at node 2, which in turn is evaluated only once target $b1$ is covered. If no test covers b_1, the ranking in MOSA is determined by the fitness function $f_1 = d(b_1)$. When tests are evaluated for the two dependent branches b_2 and b_3, the respective fitness functions will be equal to $f_1 + 1$ since the only difference from coverage of b_1 consists of a higher approach level (in this case, $+1$), while the branch distance d is the same. Since the values of f_2 and f_3 are just shifted by a constant amount, the approach level, concerning f_1, the test case ranking is the same as when considering f_1 alone. This means that objectives f_2, f_3 do not contribute to the final ranking and thus can be ignored during preference sorting.

Therefore, Panichella et al. [71] introduced the Many-Objective Sorting Algorithm with Dynamic target selection (DynaMOSA), which adds to MOSA the ability to dynamically focus the search on the subset of previously uncovered targets based on the control dependency hierarchy. The algorithm temporarily ignores uncovered targets that can only be reached by other uncovered targets placed higher in the dependency hierarchy. This is done because a population of test cases cannot cover ignored targets until their parents have been covered in the first place, which is attempted by the search focused only on the reachable parents and shall lead to a more effective consumption of the search budget. Figure 2.1 shows the nodes of the control dependence graph that DynaMOSA would consider with a given test `foo(1, 3,`

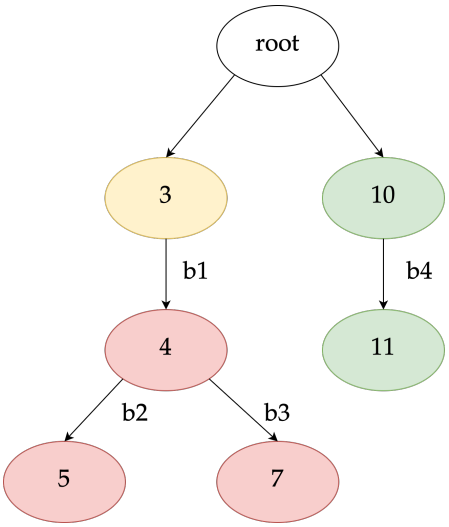
Algorithm 2 DynaMOSA [71]

Input
- U The set of coverage targets of a program
- M Population size
- G Control dependence graph of the program
- ϕ Partial mapping between CDG edges and targets

Output
- T An evolved test suite

$U* \leftarrow$ targets in U with no control dependencies
$t \leftarrow 0$
$P_t \leftarrow \text{RandomPopulation}(M)$
$archive \leftarrow \text{UpdateArchive}(P_t, \emptyset)$
$U* \leftarrow \text{UpdateTargets}(U*, G, \phi)$
while $\neg$(search_budget_consumed) **do**
  $Q_t \leftarrow \text{GenerateOffspring}(P_t)$
  $archive \leftarrow \text{UpdateArchive}(Q_t, archive)$
  $U* \leftarrow \text{UpdateTargets}(U*, G, \phi)$
  $R_t \leftarrow P_t \cup Q_t$
  $F \leftarrow \text{PreferenceSorting}(R_t, U*)$
  $P_{t+1} \leftarrow \emptyset$
  $d \leftarrow 0$
  while $|P_{t+1}| + |F_d| \leq M$ **do**
    $\text{CrowdingDistanceAssignment}(F_d, U*)$
    $P_{t+1} \leftarrow P_{t+1} \cup F_d$
    $d \leftarrow d + 1$
  end while
  $\text{Sort}(F_d)$
  $P_{t+1} \leftarrow P_{t+1} \cup F_d[1 : (M - |P_{t+1}|)]$
  $t \leftarrow t + 1$
end while
$T \leftarrow archive$
return T

Figure 2.1. Control dependency graph. DynaMOSA only considers nodes that can be reached directly from covered nodes during search.



3). Green marks the nodes covered by the test, yellow is the node that needs to be covered yet, and red nodes are temporarily ignored because the test cannot reach them at that point without covering nodes 3 first. Since DynaMOSA optimizes a subset of the targets also considered by MOSA, the former is guaranteed to be at least as efficient. In their experiments, the authors of both algorithms learned that DynaMOSA achieved significantly higher coverage than WSA and MOSA on a dataset consisting of 346 Java classes.

2.6. Automatically Generated Oracles

Traditionally, SBST is applied to generate test suites that maximize some coverage criteria, e.g., branch coverage. However, search-based techniques usually do not employ automated oracles [33]. A lack of formal specification of the behavior of a program results in generated tests that generally must be supplemented with oracles manually. Davis and Weyuker [27] introduced the term *non-testable programs*, which includes those programs which there is no test oracle for or a test oracle is impractical to implement. Thus, one cannot check the result of the computation for correctness. This includes programs that were either created to learn the outcoming result in the first place, programs that give too many results to check them all, or the developer had misunderstood the specification. To solve the problem of missing oracles, the authors introduced the so-called pseudo oracle. A pseudo-oracle is a second, independently implemented program that must comply with the specifications. The two versions need to be created by separate teams with no intermediary communication so that no misunderstandings can propagate from one group to the other. Afterward, the

Algorithm 3 PreferenceSorting(T, M) [71]

Input
T A set of candidate test cases
U Set of currently reachable objectives

Output
F Non-dominated ranking assignment

$F_0 \leftarrow \emptyset$
for $u_i \in U$ and u_i is uncovered **do**
  $t_{best} \leftarrow$ test case in T with minimum objective score for u_i
  $F_0 \leftarrow F_0 \cup \{t_{best}\}$
end for
$T \leftarrow T - F_0$
if $|F_0| >$ maximum chromosome length **then**
  $F_1 \leftarrow T$
else
  $U' \leftarrow \{g \in U : u \text{ is uncovered}\}$
  $E \leftarrow$ FastNonDominatedSort$(T, \{u \in U'\})$
  $d \leftarrow 0$
  for All non-dominated fronts in E **do**
    $F_{d+1} \leftarrow E_d$
  end for
end if
return F

Algorithm 4 UpdateArchive(T, A) [71]

Input
  T A set of candidate test cases
  A An archive
Output
  A An updated archive

for $u_i \in U$ **do**
  $t_{best} \leftarrow \emptyset$
  $best\_length \leftarrow \infty$
  if u_i already covered **then**
    $t_{best} \leftarrow$ test case in A covering u_i
    $best\_length \leftarrow$ number of statements in t_{best}
  end if
  for $t_j \in T$ **do**
    $score \leftarrow$ objective score of t_j for target u_i
    $length \leftarrow$ number of statements in t_j
    if $score == 0$ and $length \leq best\_length$ **then**
      replace t_{best} with t_j in A
      $t_{best} \leftarrow t_j$
      $best\_length \leftarrow length$
    end if
  end for
end for
return F

Algorithm 5 CrowdingDistanceAssignment(T, M) [29]

Input
  T A set of candidate test cases
  M A set of objectives

$l \leftarrow |T|$
for $i \in T$ **do**
  $T[i]_{distance} \leftarrow 0$
end for
for $m \in M$ **do**
  $T \leftarrow sort(I, m)$
  $T[1]_{distance} = I[l]_{distance} = \infty$
  for $i = 2$ to $(l - 1)$ **do**
    $T[i]_{distance} \leftarrow T[i]_{distance} + (T[i + 1].m - T[i - 1].m)$
  end for
end for

results of the computations of the original program and the pseudo oracle can be compared, and a decision on validity can be made.

Hiring a second team of developers for this task is too time-consuming and expensive for most software. And so, there have been attempts to automate this process. Korel and Al-Yami [56] described an approach that uses search techniques to find violations of assertions in code. An assertion is simply a conditional whose `false` branch is assumed not to be executed. However, genetic or symbolic search can exploit the branch to look for a program state that violates the assertion explicitly. A violation leads in most programming languages to a program crash, which is the most basic variant of an automated oracle since an unexpected crash is a bad sign in most cases.

Romano et al. [77] proposed statically selecting possible paths that cause memory access violations and searching for inputs that direct execution into those paths. It is also possible to represent arithmetic errors such as division by zero as paths so that traditional SBST metrics can optimize the search for those errors [14]. Such techniques are called *testability transformations* and were first proposed by Harman et al. [49]. Those are often source-to-source transformations intended to improve the performance of various test generation algorithms. McMinn [63] picked up the idea and proposed automatically generating pseudo-oracles for a given program. He applied testability transformations to modify the original program and develop a second version that is expected to have the same outputs as it's original but may have discrepancies. To this end, he provided two examples: floating-point arithmetic and multithreading in Java. In the former, for instance, additions between primitive floating-point types, which are somewhat imprecise in the trailing decimal digits according to the IEEE standard [46], are swapped by Java's `BigDecimal` during a transformation, e.g., the calculation $0.1+0.1+0.1$ in Java yields 0.300000000000004 instead of 0.3 (see Listing 2.2).

Listing 2.2 Comparing floating-point arithmetic in Java using double compared to `BigDecimal` [63]

```
System.out.println(0.1 + 0.1 + 0.1);
// Output: 0.30000000000000004

System.out.println(
    new BigDecimal("0.1").add(
        new BigDecimal("0.1").add(
            new BigDecimal("0.1")
        )
    )
);
// Output: 0.3
```

McMinn applied transformations to serialize and deserialize a multithreaded program in their second example. Here, methods of a class are provided with Java's `synchronize` keyword; otherwise, the keyword is removed if a method is already synchronized. `synchronize` ensures that only one thread may use the respective method simultaneously. In their evaluation, the author tries to use the oracles generated by transforming the respective SUT for a genetic-based search for input data, maximizing the discrepancy between the outputs of the original program and its pseudo-oracle. This can detect potential bugs (discrepancy) and measure their severity (size of a discrepancy). Fraser and Arkuri [42] also adopted the idea of automatically generating pseudo-oracles in their search-based test generator EvoSuite.

Transformations have not only been applied to search: the idea of checking error conditions was first mentioned in 1976 by Clarke [24] in the context of symbolic execution. Active Property Checking [45] describes usage of explicit error branches in the path constraints during DSE. By having explicit constraints on the error conditions, DSE exploration will try to negate also the error conditions, such that if there exists an input that leads to the error, it probably will be found. Other state-of-the-art DSE tools also employ this approach, e.g., Pex [85] automatically adds constraints that check references against null, divisions against zero, etc. Barr et al. [11] instrument programs with additional branches to find floating-point exceptions with symbolic execution. These additional constraints are similar to the error conditions we introduce in the Mid-level Intermediate Representation (MIR) transformations in Section 6.4.3, even though some of them do not apply to Rust due to language peculiarities, e.g., there is no concept of null pointers in safe Rust.

3

State of the Art

We give an overview of work related to ours in this chapter.

3.1. Fuzzer for Rust

Fuzzing is a widely-adopted testing method that exercises a program by automatically generating inputs in a random or heuristic way. However, fuzzing approaches require defining fuzz targets. A fuzz target represents an array of bytes as input for executing a program composed of Application Programming Interface (API) invocations [52]. API invocation chains are called fuzz drivers and usually need to be defined manually, too. Fuzzing tools can mutate the input of fuzz targets to explore different paths in SUT. Listing 3.1 shows an example of a fuzz target for the example SUT `ex_lib`.

Listing 3.1 A sample problem for fuzz target generation [52]

```
mod ex_lib {
  struct S1;
  struct S2;
  fn f1(a: i16) -> S1;
  fn f2(b: u32) -> S2;
  fn f3(c: &[u8]) -> S2;
  fn f4(s1: S1, s2: &mut S2) -> S2;
  fn f5(s2: &S2, d: &str);
}

fn example_fuzz_target_1(data: &[u8]) {
  if data.len() > 3 {return;}
  let a = to_i16(data, 0);
  let c = to_slice::<u8>(data, 2, data.len());
  let s1 = ex_lib::f1(a);
  let mut s2 = ex_lib::f3(c);
  let _ = ex_lib::f4(s1, &mut s2);
}
```

AFL++ is a reengineered fork of the popular coverage-guided fuzzer AFL by Zalewski [89]. It is an extensible community-driven open-source tool that incorporates state-of-the-art fuzzing research. The tool

mutates a set of test cases to reach previously unexplored points in the program. The test case triggering new coverage is saved as part of the test case queue [32]. Researchers can easily extend the current implementation with additional techniques and use the tool as a baseline for evaluating new ideas. *afl.rs* provides the ability to apply AFL++ to Rust.

LLVM libFuzzer is another coverage-guided evolutionary fuzzing engine. It feeds fuzzed inputs to the library under test via specific fuzzing entry points, also known as *target functions*. The fuzzer tracks which areas of the code are reached and generates mutations on the corpus of input data to maximize the coverage. Some other tools build upon the LibFuzzer and extend it with techniques like concolic testing [58, 74]. Both LLVM libFuzzer and AFL++ require manually written fuzz targets, though, i.e., a chain of invocations and a recipe where to put the generated data.

RULF is a fuzz target generator that, given the API specification of a Rust library, can generate a set of fuzz targets and seamlessly integrate them with AFL++ for fuzzing [52]. The approach leverages a sophisticated traversal algorithm to achieve high API coverage with only a tiny set of shallow fuzz targets. To construct fuzz targets, the tool builds an API dependence graph and analyses which API calls return values of the type used as a parameter for other API calls of the same SUT. Using that approach, the authors discovered several bugs in popular Rust libraries. An essential limitation of the tool is its inability to analyze and fuzz generic components.

3.2. DSE-based Test Generators

Many tools rely on DSE for the automatic generation of tests, for example CUTE, jCUTE [78], and KLEE [17].

DART [45] was the first concolic testing tool that combined dynamic test generation with random testing and model checking techniques to systematically execute as many feasible paths of a program as possible while checking each execution for various types of errors.

CUTE (Concolic Unit Testing Engine) and **jCUTE** (CUTE for Java) [78] are another pair of tools that can generate minimal tests, including input data for C as well as Java programs, respectively, exploiting DSE. The tools extend DART and generate random input data to initiate search and progress when symbolic execution fails to progress due to the limitations mentioned before. In addition, multithreaded programs can be handled that manipulate dynamic data structures using pointer operations. CUTE combines concolic execution with dynamic partial order reduction in multithreaded programs to generate both test inputs and thread schedules systematically. The tool discovered some actual bugs during the evaluation with popular open-source libraries and Java's standard library.

KLEE [17] redesigns its predecessor **EXE** [19] and is built on top of the LLVM compiler infrastructure. It uses the IR emitted during compilation. The tool performs mixed concrete/symbolic execution, models memory with bit-level accuracy, employs a variety of constraint solving optimization, and uses search heuristics to get high code coverage. In addition, for each dangerous operation (e.g., dereference, assertion), the tool tries to check whether there are values that could lead to an error in the process. In the experiments, the tool generated tests that exceeded the coverage of manually written tests and could find bugs even in such commonly used tools as parts of the GNU Coreutils. Since, even in simple programs, the number of execution states/-paths can explode, KLEE applies multiple heuristics and optimization techniques to improve its performance. *rust-verification-tools* [1] provide KLEE bindings for Rust and thus, enables usage of the tool with Rust programs.

The authors of **KLOVER** [60] describe the tool as the first to allow symbolic execution and test generation for industrial C++ programs. It builds on top of KLEE, works with the bitcode emitted by LLVM, and extends KLEE's optimizations to C++ language features, for instance, classes, objects, and LLVM intrinsics.

3.3. Search-based Test Generators

EvoSuite [37] automates the task of generating unit tests for Java by systematically producing test suites that achieve high coverage, are as small as possible, and provide assertions. The tool not only uses a search-based approach but also exploits DSE, hybrid search, and employs testability transformations [49] to improve its efficiency. The tool targets programs with no formal specification that could be used to derive oracles to test the actual behavior against the expected one. However, it uses mutation testing to produce a reduced set of assertions. The assertions highlight the relevant aspects of the current behavior to support developers in identifying regressions. EvoSuite used to apply the WS approach but has employed other GAs ever since, too, e.g., DynaMOSA. The tool sets the bar for the quality of automatically generated tests [20, 38, 39, 43, 70, 87].

SUSHI [15] is an open-source test generator for Java that combines both concolic and evolutionary testing. It does the latter by leveraging EvoSuite. The tool tries to synthesize test suites with high branch coverage. It explores the program execution space and computes the execution conditions of the program paths. Afterward, the path conditions are translated into executable evaluators that quantify the distance of a concrete state from satisfying the corresponding path condition.

[1] `https://github.com/project-oak/rust-verification-tools`

EvoObj [61] is an extension to EvoSuite. The tool tackles the problem that the effectiveness of SBST suffers greatly when generating complex object inputs due to search spaces that are neither continuous nor monotonic. EvoObj employs analysis of static control and data flow of a SUT to create *seed tests*. The seed tests can drastically increase the performance of the search by providing a more continuous and monotonic fitness landscape.

3.4. Random Testing Tools

Randoop [69] generates unit tests for Java code using feedback-directed random test generation. The technique utilizes execution feedback gathered by executing test inputs as they are created to avoid generating redundant and illegal inputs. Randoop verifies multiple basic assumptions on programs, too. For instance, programs should not crash, which is a sort of an automated oracle. It can also check contracts which the user supplies. By default, the tool implements multiple default contracts that the Java language specifies, e.g., reflexivity of `Object.equals` [33]. Randoop outputs two test suites. One of them contains contract-violating tests, while the other includes *regression* tests, which do not violate contracts but instead capture aspects of the current implementation of a SUT. Those can discover inconsistencies between two versions of the program.

4

Rust Programing Language

Rust is an increasingly popular programming language designed to be fast, efficient, and safe, whose first stable version was released in 2015. The high-level goal of the language is performance comparable to systems programming languages like C and C++ without sacrificing the safety of higher-level languages, e.g., like Java and Python.

This is a challenging problem to solve because system programming languages need to be able to control how memory is managed explicitly to be performant, which is easy for a langugage user to do incorrectly. The higher-level languages sacrifice this control for convenience, using tools such as garbage collectors to automate memory management [68]. Consider, as an example, how C and Java handle heap memory. In C, one can allocate memory on the heap by invoking the `malloc` or `calloc` functions. The memory can then be reclaimed with `free`. These functions can be easily misused, e.g., by calling `free` twice on the same pointer, which leads to undefined behavior, while forgetting to free memory leads to memory leaks. Java, and many other high-level programming languages, in contrast, use a different approach: a programmer does not need to allocate and deallocate heap memory manually since the garbage collector reclaims it at runtime now and then based on heuristics. While automatic memory management makes writing code easier and prevents memory errors, it also impacts the execution performance due to the operational overhead of determining which memory is no longer in use and freeing it.

Rust addresses this issue by the ownership system, which requires a developer to write code the way that the compiler can infer when exactly memory should be allocated and released. If the code does not satisfy the requirements, it will not compile.

4.1. Tooling

Rust comes with official package managers, which we briefly describe in the following sections.

4.1.1. Cargo

Cargo is the name of Rust's package manager and build system, which is provided out-of-the-box. It can create new packages, download de-

pendencies, and build them. It also supports more advanced features like publishing packages and running tests. Packages created and used by Cargo are known as crates. Cargo is designed to be the central tool that developers use when working with crates.

4.1.2. Rustup

While Cargo is used to manage crates, Rustup is responsible for managing Rust tools, like Cargo and the Rust compiler flavors, and any other components which have been added to the toolchain. Among other things, Rustup makes it possible to have fine-grained control over the version of Rust used. Stable and beta versions of Rust are released every six weeks, in general, while a nightly version is released every night, as the name suggests, based on the previous day's main branch in the central git repository. The beta versions are based on the latest nightly every six weeks, and the stable versions are based on the previous beta.

Features introduced in nightly versions of the compiler need to be explicitly enabled in the source code, as shown in Listing 4.1.

Listing 4.1 Enabling features in Rust

```
1 #![features(box_patterns)]
```

To develop any sort of tooling for Rust, developers must use nightly versions, although they can pin the version to a specific date if they wish. Rustup allows developers to set the version specifically for a given project or a global default.

4.2. Language Basics

The Rust language itself is unique in many ways, and we shall introduce the key points in the following sections to provide the reader a suitable knowledge base for understanding the decisions and challenges regarding the generation of test cases for Rust in the following chapters.

4.2.1. Syntax

Rust syntax is mostly in line with the C family of languages. The ubiquitous "Hello, World!" program can be written as shown in Listing 4.2.

Listing 4.2 The "Hello World" example written in Rust

```
1 println!("Hello World!");
```

The `!` sign indicates that `println` is macro, which the compiler expands during compilation. Rust's macros are way more powerful than the simple text substitution of C. During macro expansion, the relevant code is parsed into an Abstract Syntax Tree (AST) which can be

analyzed and manipulated by the macro implementation to generate code at compilation time.

In Listing 4.3, we demonstrate a struct and an enum type definition with generics. The angle brackets indicate the generics and, in this case, those are two anonymous type parameters. Enums and structs are the custom data types available in Rust. Structs package related data together and are like an object's data attributes in object-oriented languages. Enums define a type by enumerating its possible variants, which are most often accessed via pattern matching, whereas struct fields can be accessed directly.

Listing 4.3 The type definition for a point in two-dimensional space and an enum definition

```
struct Point {
  x: f64,
  y: f64
}

enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

While curly brackets indicate scope and statements end with a semicolon, Rust differs from the C family of languages in regard to its function definitions, which do not start with the return type. Typically, the types are written after the identifiers, as demonstrated in Listing 4.4. There, we compute and return a new point, placed in between the two points provided. This may look strange to those who used to seeing the `return` keyword emplaced at the end instead. It is a valid approach, however, it is considered idiomatic Rust to return by ending the function with an expression, which the compiler implicitly interpretes as a return statement.

Listing 4.4 A function to compute the point between two points in two-dimensional space

```
fn middle(p1: Point, p2: Point) -> Point {
  Point {
    x: (p1.x + p2.x) / 2,
    y: (p1.y + p2.y) / 2
  }
}
```

Structs do not have explit constructors as other languages do for classes. However, it is a convention to provide a static function called `new` that instantiates a struct.

4.2.2. *Associated Functions*

Functions can be associated with types in Rust, e.g., with enums or structs. Rust separates the definition of behavior from the definition of data. To implement behavior for an enum or a struct, we can add an `impl` block, as shown in Listing 4.5. An example of a common associated function is `new` which returns a value of the type it is associated with.

Listing 4.5 Associating behavior with the `Point` data type defined in Listing 4.3

```
impl Point {
  fn new(x: f64, y: f64) -> Self {
    Self { x, y }
  }
}

let p = Point::new(32.0f32, 42.0f32);
```

Associated functions whose first parameter is named `self` are called methods and may be invoked using the method call parameter, for instance, `x.foo()`, as well as the usual function call annotation. `self` is an instance of the type `Self`, which, in the scope of a struct, enum, or trait, is an alias for the enclosing type. Through `self`, we can access the state and methods of the instance, e.g., as in Listing 4.6.

Listing 4.6 Defining a method on `Point` data type from Listing 4.3

```
impl Point {
  fn new(x: f64, y: f64) -> Self {
    Self { x, y }
  }

  fn add(&self, other: &Self) -> Self {
    Self {
      x: self.x + other.x,
      y: self.y + other.y
    }
  }
}

let p1 = Point::new(32.0f64, 42.0f64);
let p2 = Point::new(10.0f64, 10.0f64);

let result = p1.add(&p2);

// Syntactical sugar for

let result2 = Point::add(&p1, &p2);
```

4.2.3. *Traits*

A trait describes an abstract interface that types can implement. This interface consists of associated items, which come in three variants: functions, associated types, and constants. Traits are implemented for specific types through separate `impl` blocks. Their functions can have default implementations, which implementing types can exploit if they do not provide custom implementations. Otherwise, an unimplemented trait's function body indicates that the implementor must override it. Similarly, associated constants may omit the equals sign and expression to indicate implementations must define the constant value. Associated types must never define the type; the type may only be specified in an implementation. As an example, a trait could be defined for implementing a method that generates a string from an instance of a type, as shown in Listing 4.7.

Listing 4.7 Trait definition and implementation for the `Point` data type from Listing 4.3

```
trait ToString {
  fn to_string(&self) -> String;
}

impl ToString for Point {
  fn to_string(&self) -> String {
    format!("({}, {})", self.x, self.y)
  }
}

let p = Point::new(32.0f64, 42.0f64);
let textual_repr = p.to_string();
```

The strong macro system also allows to automatically generate implementations of some traits, e.g., the `Clone` trait, which provides the ability to clone an instance. To this end, we annotate a struct or an enum with the macro `#[derive( ... )]` and list the traits to derive, as shown in Listing 4.8. Deriving the trait requires that all attributes of the `Point` structure also implement `Clone` (which `f64` does) because the generated implementation invokes `.clone()` on each attribute.

Listing 4.8 The compiler automatically generates an implementation of the `Clone` trait for the `Point` data type

```
#[derive(Clone)]
struct Point {
  x: f64,
  y: f64
}

fn main() {
```

```
    let p1 = Point::new(32.0f64, 42.0f64);
    // We can now clone the instance
    let p2 = p1.clone();
}
```

4.2.4. Generics and Trait Objects

Rust supports generic types and allows for shared behavior via traits. Generics parametrize data structures and functions, such that different types can reuse same code. This improves usability and helps finding type errors statically. The language provides static and dynamic dispatch. The former is realized by means of monomorphization, i.e., for each type a generic implementation is used with, the compiler generates a concrete implementation of it and replaces the call sites with calls to the specialized functions. As a consequence, the final binary will be slightly larger due to potentially many copies of the same code, but calling the functions will not result in any runtime dispatch overhead. Moreover, due to static type information, the compiler can inline the code, which generally leads to better performance. Listing 4.9 demonstrates a generic data type that uses static dispatch.

Listing 4.9 A data type with static dispatch via monomorphization

```
struct Pair<L, R> {
  left: L,
  right: R
}

impl<L, R> Pair<L, R> {
  pub fn of(left: L, right: R) -> Self {
    Pair { left, right }
  }

  pub fn left(&self) -> &L {
    &self.left
  }

  pub fn right(&self) -> &R {
    &self.right
  }
}
```

With dynamic dispatch, trait objects come into play. Those are usual instances of any type that implements the given trait, where the precise type can only be known at runtime. The methods of the trait can be called on a trait object via the *vtable*, which is created an managed by the compiler. A function that takes trait objects is not specialized to each type that implements the trait: only one copy of the code is

generated, resulting in less code bloat. However, this comes at the cost of requiring slower virtual function calls and effectively inhibiting any chance of inlining and related optimizations from occurring. When using trait objects, they must be packed into a data type with a known compile-time size, e.g., a `Box`, a pointer into the heap, as demonstrated in Listing 4.10.

Listing 4.10 A data type with dynamic dispatch

```
struct Pair {
  left: Box<dyn Any>,
  right: Box<dyn Any>
}


impl Pair {
  pub fn of(left: Box<dyn Any>, right: Box<dyn Any>)
    -> Self {
    Pair { left, right }
  }

  pub fn left(&self) -> &Box<dyn Any> {
    &self.left
  }

  pub fn right(&self) -> &Box<dyn Any> {
    &self.right
  }
}
```

With the datatype definition from Listing 4.9, we can now create `Pair` instances with different types as shown in Section 4.2.4 using monomorphization, e.g., `String` and `u32` or two times `usize`. The same code is reused, and the compiler can statically check whether the pair instance uses values of correct type:

```
fn main() {
  let pair: Pair<String, u32> = Pair::of(
    "Hello world".to_string(), 42u32
  );
  let left_value: &str = pair.left();

  let another_pair: Pair<usize, usize> = Pair::of(
    0usize, 1usize
  );
}
```

It is also possible to put constraints, called trait bounds in Rust, on the generic type parameters. For instance, we can define a textual rep-

resentation of a pair. The `Display` trait from the standard library provides the `fmt` method, which uses the {} print marker. As shown in Section 4.2.4, for a type to implement the `Display` trait, its attributes must implement the trait, too, since their `fmt` method will be called recursively. Hence, we tell that the left and right hand side type parameters used with the `Pair` struct must implement the `Display` trait. The effect of bounding is that generic instances are allowed to access the methods of traits specified in the bounds. Under the hood, the `println!` macro in Line 23 calls the `fmt` method of the `Pair` instance:

```
1  struct Pair<L: Display, R: Display> {
2      left: L,
3      right: R
4  }
5
6  impl<L: Display, R: Display> Pair<L, R> {
7    pub fn of(left: L, right: R) -> Self {
8      Pair { left, right }
9    }
10 }
11
12 impl<L: Display, R: Display> Display for Pair<L, R> {
13   fn fmt(&self, f: &mut std::fmt::Formatter)
14       -> std::fmt::Result {
15     write!(f, "({}, {})", self.left, self.right)
16   }
17 }
18
19 fn main() {
20   let pair: Pair<String, u32> = Pair::of(
21     "Hello world".to_string(), 42u32
22   );
23   println!("{}", pair);
24   // Output: (Hello world, 42)
25 }
```

There is no inheritance in Rust, as known from object-oriented languages. Common behavior can be defined via traits, which can have super-traits, as shown in Listing 4.11 for the `TrieAtom` trait. `TrieAtom` is a blanket trait, which means that types that implement its super trait automatically implement it, too, from the compiler's point of view.

Listing 4.11 An example trait from the *trying* crate

```
1  pub trait TrieAtom: Copy + Default + PartialEq + Ord {}
```

Besides structs, it is also possible to parametrize methods and functions using generics. A generic method or function has a type parameter that is inferred from the values passed as parameters.

Listing 4.12 Variants of defining a generic function

```
fn make_pair<L: Display, R: Display>(left: L, right: R)
    -> Pair<L, R> {
  Pair::of(left, right)
}

// Alternative syntax for monomorphism,
// useful when the definition grows large
fn make_pair<L, R>(left: L, right: R) -> Pair<L, R>
where L: Display, R: Display {
  Pair::of(left, right)
}

fn main() {
  let pair: Pair<String, String> = make_pair(
    "Hello".to_string(),
    "world".to_string()
  );
  let another_pair: Pair<usize, u32> = make_pair(
    4usize, 2u32
  );
}
```

The function `make_pair` creates a generic `Pair` instance. Then, it is used again to create a pair of two strings and a pair of a `usize` and a `u32`.

Traits may also contain additional type parameters. Other traits may constrain these type parameters and so on; see Listing 4.13. Usually, the more constraints, the more difficult it is to find correct types and automatically generate test cases that use those features and do compile.

Listing 4.13 Type parameters can be specified for a trait to make it generic. These appear after the trait name, using the same syntax used in generic functions

```
trait PrintableSeq<T: Display> {
  fn len(&self) -> usize;
  fn at(&self, pos: usize) -> T;
  fn iter<F>(&self, f: F) -> where F: Fn(T);
}
```

4.2.5. Ownership and Borrowing

As mentioned at the beginning, Rust's approach to memory management is different from the traditional one of C or C++, where memory is managed manually through functions like `malloc` and `free`, or by letting the garbage collector manage the resources. In Rust, each value has a variable that owns the value. There can only be one owner of

a value at a time, and once the owner goes out of scope, the value is deallocated.

Listing 4.14 Heap data owned by binding

```
{
  // The String data on the heap is created
  // and set to be owned by binding a
  let a = String::from("content");
}
// The scope in which a is active is over,
// and a is now no longer valid. At the same
// time the string data on the heap is freed
```

In cases where the value needs to outlive a given scope, it can be moved to a new owner in a different scope. This happens, for example, when a value is passed as a parameter to a function, or on assignment, with the caveat that some values are so cheaply copied that a move is never necessary.

Listing 4.15 Transferring ownership

```
let b;
{
    let a = String::from("content");
    b = a;

    println!("The content is: {}", a);
    // Trying to use a after transferring the
    // ownership results in a compilation error
}
```

Moving values around would quickly become burdensome in cases when we want to reuse a value, e.g., after passing it to a function invocation. In Rust, we can also reference variables. References can be immutable (read-only) or mutable, meaning that the reference owner has write access to the underlying value. Creating a reference is also known as borrowing in Rust, a fundamental interaction aspect in the ownership model. The reference owner cannot modify the value by default when borrowing a value. It can only be borrowed mutably if it is not referenced somewhere else for the duration of the borrow. On the other hand, a value can be borrowed immutably multiple times, as there is no possibility of a race condition when the underlying cannot be modified.

These rules make it possible for the compiler to infer when resources can be deallocated, guarantee memory safety, and make it impossible to create structures that are not tree-formed without an additional level of indirection. For example, implementing a linked list with these restrictions would be impossible since a list node is a recursive data structure. That is because the ownership system is an over-approximation

of memory safety; sometimes, the compiler can reject completely valid and safe code. This is why some parts of the standard library are implemented using the `unsafe` keyword, which allows the programmer, among other things, to bypass these rules. In practice, this means that memory bugs still happen in Rust but the ownership system prevents them from originating in safe Rust.

Listing 4.16 Transferring the ownership to a method

```
fn main() {
    let mut message = String::from("Hello");
    print(message); // Value moved here
    // Compile error, value borrowed here after move
    message.push_str(" world!");
}

fn print(message: String) {
  // message gets dropped when the execution
  // goes out of scope
  println!("{}", message);
}
```

The problem with the ownership model described above is that if we want to continue working with the moved value after a function call, such as in Listing 4.16, the called function must return the value to the call site since, in the example, the string moved into the function `print`. Instead, we can pass a reference to the value as an argument to the invoked function, which is like a pointer to the value, so the function can follow the address and access the value. Unlike a pointer in C, a reference in safe Rust is guaranteed to point to the correct memory. As shown in Listing 4.17, a reference to a value can be passed using `&`. The syntax `&message` provides the ability to reference the string value, i.e., borrow it. Since the reference does not own the value, the actual value will not be deallocated when the reference is no longer alive.

Listing 4.17 Transferring the ownership to a method

```
fn main() {
    let mut message = String::from("Hello");
    // A reference to message
    // it passed to the function call
    print(&message);
    // Variable message still owns the value
    // after the call
    message.push_str(" world!");
}

// The function borrows the value of message
fn print(message: &String) {
```

```
  println!("{}", message);
}
```

Whether owned or borrowed, values cannot be changed by default, as this is prevented by the compiler. If this is still necessary, one has to explicitly specify it with the keyword `mut`, as done in Listing 4.18.

Listing 4.18 Transferring the ownership to a method

```
fn main() {
    let mut text = String::from("Hello");
    extend_msg(&mut text);
    println!("{}", &text);
    // Output: Hello world!
}

fn extend_text(text: &mut String) {
    text.push_str(" world!");
}
```

Thus, the restrictive ownership model has particular implications for the generated tests, i.e., we cannot use variables arbitrarily and need to observe their ownership state.

4.2.6. Lifetimes

A *lifetime* is an import aspect of the borrowing and ensures that all borrows are valid. A variable's lifetime begins when it is created and ends when it is destroyed. When we borrow a variable with `&`, the lifetime of the borrow begins. Its end is determined by where the reference is still used. Take, for instance, the small program in Listing 4.19.

Listing 4.19 An example program that explicitly declares lifetimes

```
struct Int {
  x: i32
}

impl Int {
  fn consume(self) {
    // ...
  }
}

trait Value {
  type T;
  fn value(&self) -> &Self::T;
}

impl Value for Int {
```

```
17    type T = i32;
18
19    fn value(&self) -> &i32 {
20      &self.x
21    }
22  }
23
24  struct ValuePair<'a, V, C>
25  where V: Value<T = C> {
26    left: &'a V,
27    right: &'a V
28  }
29
30  impl<'a, V, C> ValuePair<'a, V, C>
31  where V: Value<T = C> {
32    fn left(&self) -> &'a V {
33      &self.left
34    }
35
36    // ...
37  }
38
39  fn foo() {
40    let mut i1 = Int {x: 42};
41    let mut i2 = Int {x: 101};
42
43    let pair = ValuePair {
44      left: &i1,
45      right: &i2
46    };
47
48    let i2_value = i2.consume();
49  }
```

In the example, we declare a generic struct `ValuePair`, which holds two references of a type that implements the trait `Value` with the associated type `T` being some generic type `C`. The struct declares a named lifetime `'a`. Lifetime names always begin with an apostrophe. There is also a struct `Int`, which implements the trait and thus, can be used in combination with `ValuePair`. Within the function `foo`, we instantiate a pair which effectively borrows the two `Int` instances `i1` and `i2`. Afterwad, in Line 48, we invoke the function `consume`, which, as the name suggests, consumes the value of the callee, which theoretically clashes with the `i2` being borrowed by the pair instantiation. The code compiles, though, since the pair is no longer used after its initialization; thus, the borrow checker considers the lifetime of the borrow ended.

However, the situation changes if we insert another statement on the pair at the end of `foo`, for instance:

```
fn foo() {
  // ...

  let left_value = pair.left();
}
```

Since we call a function on the `ValuePair` instance, the lifetime of the instance gets extended until that point, resulting in a conflict in Line 48 because we try to move a borrowed value. So why does it matter to us? When we generate and recombine test cases for crates that heavily make use of references in custom datatypes, we will need to face situations like that one in the small example, which, when not handled properly, would lead to many uncompilable test cases that are of no value. In Section 5.1, we describe a basic strategy which RustyUnit applies to mitigate most common forms of this "issue" but it covers by far not all possible cases. We call it "issue" since it poses a practical challenge for our approach but in fact, borrow checker is one of Rust's outstanding features.

4.3. Overview of the Compiler

As with most compilers, the Rust compiler (*rustc*) goes through several phases, using multiple IRs to facilitate computations. Working directly with the source code is highly inconvenient and error-prone. Source code is designed to be human-friendly while at the same time being unambiguous, but it's less convenient for analysis, e.g., type checking.

Instead, most compilers, including *rustc*, build some IRs out of the source code, which is easier to process. rustc has a few IRs, each optimized for different purposes:

- Token stream: The lexer produces a stream of tokens directly from the source code. This stream is easier for the parser to deal with than raw text.
- AST: The abstract syntax tree is built from the stream of tokens. It represents almost exactly what a user wrote. It helps to do some syntactic sanity checking (e.g., checking that a type is expected where the user wrote one).
- High-level Intermediate Representation (HIR): This is a desugared and compiler-friendly representation of the AST that is generated after parsing, macro expansion, and name resolution. It is still close to what the user wrote syntactically, but it includes implicit information such as elided lifetimes and generated implementations. Also, some expression forms have been converted to

simpler ones, e.g., `for` loops are converted to a more basic `loop`. This IR is amenable to type checking.

- Typed HIR (THIR): This is an intermediate between HIR and MIR. It's similar to HIR but it is fully typed and a bit more desugared.

- MIR: This IR is a Control Flow Graph (CFG). It shows the basic blocks of a program and how control flow connects them. A basic block contains simple typed statements (e.g., assignments, primitive computations, etc.) and control flow pointers to other basic blocks. This representation is used for borrow checking and other important data flow-based checks, such as exposing uninitialized values.

- LLVM IR: This is a standard form of all input to the LLVM compiler. LLVM IR is a typed assembly language with lots of annotations. It is designed to be easy for other compilers to emit and rich enough for LLVM to run optimizations.

4.3.1. High-Level Intermediate Representation

Many parts of the HIR resemble Rust surface syntax quite closely, with the exception that some of Rust's expression forms have been desugared away.

For the analysis of available data types and their functions, we need to fetch as much information as possible which is provided by the compiler. The HIR is perfectly suitable for this purpose because the compiler fills implicit type information but the representation still keeps all language elements that are relevant for us. Only intraprocedural elements are desugared, for instance, `for` loops are converted into a `loop` and do not appear in the HIR.

Listing 4.20 An example Rust program that we lower to its HIR

```
struct IntVec {
  vec: Vec<i32>
}

impl IntVec {
  pub fn foo(&mut self) {
    for n in self.vec {
      // Does something
    }
  }
}
```

Listings 4.20 and 4.21 compare an example program representation before and after lowering to HIR. The main differences are the desug-

ared `for` loop and the filled-in type declaration of the `self` parameter of the `foo` method.

Listing 4.21 HIR of the code shown in Listing 4.20

```
struct IntVec {
  vec: Vec<i32>,
}

impl IntVec {
  pub fn foo(self: &mut Self) -> () {
    {
      let _t = match #[lang = "into_iter"](self.vec) {
        mut iter => loop {
          match #[lang = "next"](&mut iter) {
            #[lang = "None"] {} => break,
            #[lang = "Some"] { 0: n } => {
              {
                // Does something
              };
            }
          }
        },
      };
      _t
    }
  }
}
```

The next lowering step, i.e., THIR, similarly to MIR, only contains function bodies, i.e., executable code. Consequently, THIR has no representation for items such as `struct` or `trait` and is therefore not suitable for this type of analysis.

4.3.2. *Mid-Level Intermediate Representation*

MIR is based on the CFG of a function body and is fully typed. The MIR for a given function body consists of a series of basic blocks. A basic block is an atomic unit of the program's CFG, is guaranteed to execute entirely, and is made up of a sequence of statements followed, optionally, by a terminator, which corresponds to an edge in the CFG. Being very much internal to the Rust compiler, MIR does not have a stable definition and undergoes frequent changes. There is no stable surface syntax for MIR, only one intended for debugging, which we will use here to show examples of analysis and instrumentation. Instead, it is defined as a collection of data structures. MIR has enough information to make our program instrumentation feasible, though its

unstable nature makes the consumption of MIR challenging to maintain across different toolchain versions.

At this level, each executable piece of code, e.g., a function or a method, is represented as a `Body` object, which contains information about locals and basic blocks. Locals are memory allocations on the stack (conceptually, at least), such as function arguments, local variables, and temporaries. These are identified by an index, written with a leading underscore, like `_1`. A special local (`_0`) is allocated to store the return value.

Basic blocks consist of statements, which are, in terms of MIR, actions with one successor. The statements of a basic block are mostly definitions of locals, e.g., atomic computations, which the corresponding terminator uses later. A terminator is an action with potentially multiple successors and is always placed at the end of a basic block. A terminator can be, for instance, a `Call` (function call), a `Return` (returns the `_0` local), a `Goto` jump (jump to a given block number), a `SwitchInt`, or an `Assert`. RustyUnit makes use mainly of `SwitchInt` and `Assert` terminators.

`SwitchInt` results from lowering `if` and `loop` conditions and `match` expressions. The value of its operand always evaluates to an integer; the execution jumps depending on its value to one of the targets. There is also a dedicated *otherwise* value for all other cases. Each possible value (e.g., 0 and 1 for *false* and *true*, respectively, in if conditions) has a corresponding basic block which the execution jumps to. Assume that there are two functions, `cheeky_itoa(x: i32)` and `bold_itoa(x: i32)`, which both return a string (more precisely, a reference to a static string value) based on the input number, as shown in Listing 4.22.

Listing 4.22 HIR of the code in Listing 4.20

```
fn cheeky_itoa(x: i32) -> &'static str {
  match x {
    0 => "There is nothing here,
      only the infinite emptiness",
    42 => "The answer to everything",
    _ => "Must be some huge value"
  }
}

fn bold_itoa(x: i32) -> &'static str {
  if x > 42 {
    "Must be some huge number"
  } else {
    "Not that huge, I'd say"
  }
}
```

As mentioned earlier, conditions like those in loops, `match` and `if` expressions are converted to `SwitchInt` terminators. In Listing 4.23, the MIR for the function `cheeky_itoa` is shown in its debug form. It has been cleaned up for simplicity and usually still includes many debugging comments on the individual statements and terminators. Only two locals are given, the return value `_0` of type `&str` and the function parameter `_1` of type `i32`.

Listing 4.23 MIR of the `cheeky_itoa` function

```
fn cheeky_itoa(_1: i32) -> &str {
  let mut _0: &str;

  bb0: {
    SwitchInt(_1) -> [
      0_i32: bb2,
      42_i32: bb3,
      otherwise: bb1
    ];
  }

  bb1: {
    _0 = const "Must be some huge value";
    goto -> bb4;
  }

  bb2: {
    _0 = const "There is nothing here,
      only the infinite emptiness";
    goto -> bb4;
  }

  bb3: {
    _0 = const "The answer to everything";
    goto -> bb4;
  }

  bb4: {
    return;
  }
}
```

We can see that the entry block, which is always `bb0`, consists only of the `SwitchInt` terminator, which, based on the value of the function parameter `_1`, directs execution to either block number 2 (if `_1` is 0) or 3 (if `_1` is 42). If neither case matches, the execution jumps to block 1. For the decision, the values, which are always integers, are directly compared. Each of the following three blocks defines the return local and

points to the last block (`_4`), which returns `_0` to the caller. The MIR of the `bold_itoa` function is structured similarly (Listing 4.24). The entry block compares the parameter value with the constant 42 in Line 8 and stores the boolean result, which technically is an integer (0 or 1 for false and true, respectively), into the local `_2`. The `SwitchInt` decides again, based on the value of its operand, which block the execution must jump to.

Listing 4.24 MIR of the `bold_itoa` function

```
1  fn bold_itoa(_1: i32) -> &str {
2    let mut _0: &str;
3    let mut _2: bool;
4    let mut _3: i32;
5
6    bb0: {
7      _3 = _1;
8      _2 = Gt(move _3, const 42_i32);
9      SwitchInt(move _2) -> [false: bb2, otherwise: bb1];
10   }
11
12   bb1: {
13     _0 = const "Must be some huge number";
14     goto -> bb3;
15   }
16
17   bb2: {
18     _0 = const "Not that huge, I'd say";
19     goto -> bb3;
20   }
21
22   bb3: {
23     return;
24   }
25 }
```

As one can see, the terminators described are essentially responsible for determining the direction of execution. We will take advantage of this property and instrument these terminators in tested Rust programs to track program execution triggered by tests generated from RustyUnit. Thus, we will learn which tests cover which paths.

5

Search-based Unit Test Generation in Rust

To evolve test suites that optimize the coverage criterion, we use the DynaMOSA genetic algorithm. In this chapter, we describe the adoptation of the GA to the problem of test generation for Rust programs regarding the representation of chromosomes in Section 5.1, genetic operations in Section 5.2, the fitness function in Section 5.3, the applied testability transformations in Section 5.4, and seeding of initial population in Section 5.5.

5.1. Problem representation

According to McMinn [64], an encoding of the solution should be modeled in a way that similar solutions are also neighbors in the represented search space. This makes it easy to continue the search from one to a similar solution by applying simple representation modifications. A chromosome in a population typically represents an entire test suite in single-objective GAs for test generation [21, 36]. However, Multi- and Many-Objective Algorithms (MOAs) define an individual as a single test case [71]. Chromosome representation directly impacts how the search operators work. For instance, recombining two test suites is trivial by recombining only the sequences of their test cases at specific cut points since individual test cases are independent of each other [42]. Recombining test cases is more complicated because it means recombining their sequences of statements, which are, however, very much interdependent. For example, a statement could instantiate an object by invoking an appropriate constructor, followed by a function invocation on that object.

RustyUnit models a chromosome as a test case, a sequence of statements or program calls that execute parts of the SUT to reach and cover a particular objective. We also also need to take into account that Rust programs are not just procedures but have a certain class-like structure. Test cases only need to call functions with certain input data to achieve high coverage within a procedure-like environment. However, instances of structs can have states that direct accesses or method invocations can change. The control flow of a method may depend on the internal state of an object. A certain statement call sequence may

be essential to achieve high code coverage. For the generation of test cases with a genetic algorithm, we implement already known ideas for representing genetic individuals [7, 44, 86]. Similar to Fraser's and Arcuri's [36] definition, each statement s_i in a test case is a value $v(s_i)$, which has a type $\tau(v(s_i)) \in \mathcal{T}$, where $\mathcal{T}$ is the finite set of types. There can be five different types of statements:

- **Primitive statements** represent numeric variables, e.g., `let v = 42`. The primitive variable defines the value and type of the statement.
- **Struct initializations** generate instances of a given struct, e.g., `let b = Book { name: "The Hobbit" }`. The object constructed in the statement defines the value and statement's type. A struct instantiation can have parameters whose values are assigned out of the set $\{v(s_k) \mid 0 \leq k < i\}$.
- **Enum initializations** generate instances of a given enum, e.g., `let opt: Option<i32> = None;`. The enum instance defines the value and statement's type. An enum instantiation can have parameters whose values are assigned out of the set $\{v(s_k) \mid 0 \leq k < i\}$.
- **Field statements** access member variables of objects, e.g., `let b = a.x`. The member variable defines the value and the field's statement type. The source of the member variable, i.e., `a`, must be part of the set $\{v(s_k) \mid 0 \leq k < i\}$. Since unit tests are usually contained in the same module as the unit under test, tests can also legally access private fields.
- **Associative function statements** invoke associative functions of datatypes, e.g., `let b = a.len()`. The owner of the function (if non-static) and all of the parameters must be values in $\{v(s_k) \mid 0 \leq k < i\}$. The return value determines the statement's value and type. In the following, we refer to associative functions, too, when we talk about functions, unless otherwise stated.
- **Function statements** invoke loose functions, i.e., functions that are not associated with any datatype, for instance, `let a = foo()`. The parameters of the function must be values in $\{v(s_k) \mid 0 \leq k < i\}$. The return value determines the statement's value and type.

The collection of available structs and enums, their fields, associative functions, and loose functions are so-called test cluster [36]. The size of a test suite and individual tests is dynamic and can change (almost) arbitrarily. Since RustyUnit does not produce test oracles for the generated tests, the size value should have an upper limit so that the tests keep comprehendably for a human. Of course, it also makes sense not to let a test become completely blank.

Each statement that does not have the default return value `()` defines a new variable. However, a generated test cannot be composed arbitrarily of the above building blocks. Each test is subject to the same constraints as regular Rust programs, which the compiler checks [86].

Functions and fields in a test are not limited to just the parts of the module under the test since complex sequences of calls might be necessary to define some arguments [44]. Fraser and Zeller [44] describe in their work on the mutation-based generation of tests for Java classes an abstract construct that quickly illustrates the construction steps of a unit test. We adopt their concepts to Rust and implement them in RustyUnit. Let $parameters(M)$ be a function that returns a list of data types of the parameters of a function M, including the caller in the case of a non-static associative function. Furthermore, let $instances(t)$ be a function that returns the set of instances that have already been created in a test t and $type(i)$ be a function that returns the datatype of an instance. A function is a generator of a datatype D if it returns a value of type D or a (partially) generic type that can be D at runtime. For instance, `Option<i32>` generally can be derived from a function that is declared to return `Option<T>`.

Furthermore, any struct/enum initializer of type D is also a generator of D. Let $generators(M, D)$ be a function that returns the set of generators for the datatype D from the set of functions and initializers M. RustyUnit generates random test cases following Algorithm 6, for instance, to create an initial population. It inserts a randomly selected function s from M into an initially empty test case. If s has parameters, RustyUnit instantiates appropriate arguments using *GenObject* Algorithm 7 and inserts the generated statements before s in t. In the subsequent phase, the test case is populated until size l by generating random statements from M that accept at least one datatype d from t as parameter. Thus, the variables already present in t are reused to build a cohesive test case. Arguments for the remaining parameters that cannot be satisfied with values from t are generated again using *GenObject*.

Algorithm 7 describes how *GenObject* generates an instance of type d from set of generators M for a test case t. It randomly selects a generator s for d that does not require any parameter which G already contains and recursively generates arguments for parameters of s that cannot be satisfied with variables from t. Finally, RustyUnit inserts s as a statement into t with the appropriate arguments from t.

To this end, corresponding calls that have return values of suitable data types are inserted recursively (Algorithm 7) if instances of such a data type are not already defined in the test case, or none of them are usable according to the ownership model of Rust. However, Fraser and Zeller [44] suggest reusing existing objects only with a certain likelihood to increase diversity in generated test cases. Their experiments showed that a probability of 90% percent was most effective. Rust's type system makes reusing of previously defined objects more com-

Algorithm 6 GenTest(M, l, p_{local})

Input
- M Set of all functions and initializers
- l Desired max length of a test case
- p_{local} Probability to use local available instances as parameters

Output
- t Randomly generated test

$t \leftarrow \langle\rangle$
$s \leftarrow$ randomly select an element from M
for $p \in parameters(s)$ **do**
  $t \leftarrow GenObject(p, \{\}, M, t)$
end for
$t \leftarrow t.s$
while $|t| < l$ **do**
  $d \leftarrow$ randomly select datatype in $datatypes(t)$
  $M' \leftarrow \{m \mid m \in M \wedge d \in parameters(m)\}$
  $s \leftarrow$ randomly select function from M'
  for $p \in parameters(s)$ **do**
    $V \leftarrow \{v \mid v \in instances(t) \wedge v$ is usable by $s\}$
    if $random \geq p_{local} \vee V$ is empty **then**
      $t \leftarrow GenObject(p, \{\}, M, t)$
    end if
  end for
  $s \leftarrow$ set parameters of s to values from t
  $t \leftarrow t.s$
end while
return t

plicated, though, because it requires analysis of the extent to which a given variable is still usable in a given test case. A statement s may only use an already defined variable if this complies to ownership rules. More precisely, RustyUnit handles instances in a test as follows: let t be a test case, and $pos(s)$ be a function that returns the position of a statement s in the sequence of statements of t. Let o be an object defined by a statement gen. Let s be a statement that uses o. Then t is a valid test case only if the following rules are satisfied:

1. $pos(gen) < pos(s)$, AND
2. if o is not used by any statement $s' \in t$, o may be freely consumed and borrowed by s, AND
3. if o is borrowed from a statement $s' \in t$, o may also be borrowed from s at position p_{borrow} with $pos(gen) < p_{borrow}$ or consumed at position $p_{consume}$ with $pos(s') < p_{consume} < |t|$, AND

Algorithm 7 GenObject(d, G, M, t)

Input
- d Datatype of desired object
- G Set of datatypes already attempting to generate
- M Set of all method, constructors and functions
- t Test case

Output
- t Test case extended with an instance of d

$M' \leftarrow generators(M, d)$
$s \leftarrow$ randomly select an element from M'
for $p \in parameters(s)$ **do**
  $V \leftarrow \{v \mid v \in instances(t) \wedge v$ is usable by $s\}$
  if $random \geq p_{local} \vee V$ is empty **then**
    $G' \leftarrow G \cup \{d\}$
    $t \leftarrow$ GenObject(p, G', M, t)
  end if
end for
$s \leftarrow$ set parameters of s to values from t
$t \leftarrow t.s$
return t

4. if o is consumed by a statement $s' \in t$, o may now be borrowed only from s' at position p_{borrow} with $pos(gen) < p_{borrow} < pos(s)$.

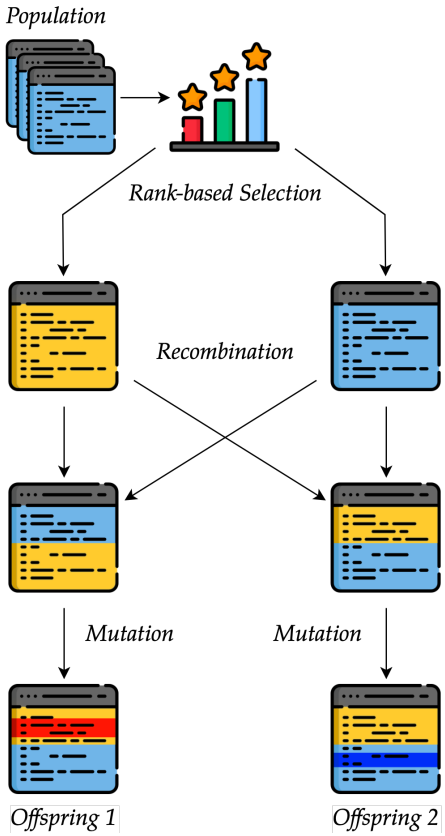
These rules must also be strictly followed when applying genetic operators during the search so that evolved test cases still represent valid Rust code. Since the prototype of RustyUnit does not provide the ability to construct all possible types, e.g., closures or trait objects, due to technical limitations, `GenObject` may fail to generate an instance of the appropriate type. In this case, RustyUnit skips the particular function it tried to generate the argument for. Obviously, this might lead to worse coverage results, but a complete analysis of all possible cases is not trivial and cannot be done within the scope of this work. Since the limitations are mainly an engineering issue, RustyUnit can be extended to support further language features in the future.

5.2. Search Operators

RustyUnit evolves a population by repeated selection of test cases and applying crossover and mutation operators according to the configured probabilities. The key is to guide the search toward solutions with better fitness values. To this end, the tool applies rank selection. Once the set of parents has been selected, recombination can take place to form the next generation. It is applied to individuals with a certain probability. At this point, the offspring might require postprocessing.

The dependencies of their statements need to be considered, i.e., possibly missing arguments must be generated to repair the tests. Figure 5.1 illustrates the steps RustyUnit applies to evolve a test suite.

Figure 5.1. Abstract overview of the steps RustyUnit applies to evolve a population



After selection and crossover have been applied, the offspring can mutate. There are different ways of mutating a sequence-based test case, e.g., delete, insert, or modify a statement [44]. Modifying a statement includes changing the parameters of a function or replacing the function invocation itself with another one. The rules defined in Section 5.1 determine when the offspring is valid with respect to the Rust compiler, e.g., replacing the function call `fn foo(&self)` by `fn bar(self)` is not allowed if the callee of the function is also used by another statement in some way later in the sequence because `bar` moves the its argument.

5.2.1. *Crossover*

RustyUnit applies the simplest type of crossover, the single-point crossover, which cuts two test cases at a random position and generates two new ones by swapping their subsequences [44]. Sice the tool models chromosomes in the natural way, i.e., as a sequence of statements, when re-

combining a pair of chromosomes, attention must be paid to the data dependencies between individual statements. For instance, there is a dependency between a function call and its arguments, which we always declare and initialize in separate statements; that is, we use the Static Single Assignment (SSA) form for the generated tests. If the intersection of a recombination hits such a dependency between statements, the offspring will not be accepted by the compiler in most cases and needs postprocessing. Tonella [86] has proposed two solutions to this problem:

- Missing dependencies of statements can be added by generating, as with the mutation operator. If certain definitions become redundant because their later use has been pushed into another choromosome by recombination, they can be deleted.
- Dependencies from a part of a chromosome shifted by recombination are also shifted to another chromosome regardless of the crossover point, provided they are no longer needed in the original chromosome, otherwise they could be copied.

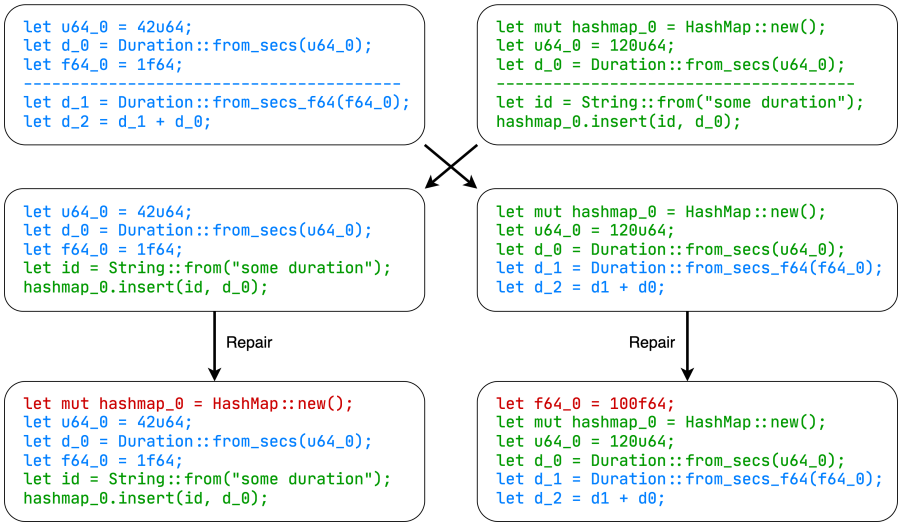
For recombination, RustyUnit exploits the first option: in case a recombined test case misses some data dependencies, the tool regenerates them afterwards using the `GenObject` algorithm, so that the test case becomes compilable again. Figure 5.2 illustrates an example for such scenario. A random intersection in the two test cases blue and green is determined. By recombining the statements, two new test cases are created, both of which, however, cannot be compiled as is since both the left (`hashmap_0`) and the right test case (`f64_0`) utilize objects that are not defined in the respective tests after all. To recover, RustyUnit performs a postprocessing step to repair the broken tests (marked red).

The way RustyUnit applies recombination can make some variables redundant, such as the `hashmap_0` instance in the right recombined test case. Such statements add no value to a test case and make it unnecessary longer and less comprehensible. We cannot not know in general which statements are useful, i.e., increase coverage, and which are not, unless we remove them one by one from a test case and re-run it. However, we apply a simple heuristic: we remove statements whose value is not used by any other statement in a test case in case a statement is primitive or the item it invokes is declared outside of the SUT.

5.2.2. *Mutation*

RustyUnit models a test case as sequence of statements. Accordingly, we need to specify possible atomic operations for a mutation. To this end, we adopt Fraser's and Arcuri's [44] definition of the mutation for Rust. Thus, RustyUnit can perform the following operations when mutating a test:

Figure 5.2. Crossover between two test cases with postprocessing



Insert a statement A given test case t is extended with a probability σ by a new statement, which is inserted at a random position i in t. A new statement can only be inserted if $l(t) < L$, i.e., if the test length does not exceed a defined maximum. For each insertion, with probability 1/3 a random call out of the pool of callables is inserted. Any parameters of the selected invocation are either reused out of the set $\{v(s_k) \mid 0 \leq k < i\}$ or randomly generated. Each parameter p selected from $\{v(s_k) \mid 0 \leq k < i\}$ must be usable by the invocation according to the rules in Section 5.1. Multiple statements can also be inserted repeatedly; this can happen with probability $p(n) = 0.5^n$ with n being the number of calls inserted so far. That is, after each insertion, a random decision is made whether to repeat the mutation [86].

An example in Listing 5.1 demonstrates how two additional function calls `push` and `len` on an existing vector instance have been inserted in Lines 18 and 19, respectively. Since `push` requires one argument, a 32-bit integer in this case, it has also been generated.

Listing 5.1 The functions `push` and `len` have been selected randomly to be invoked on *vec_0*. An appropriate argument has been generated for the second *push*, too

```
1  #[test]
2  fn test() {
3    let usize_0 = 32usize;
4    let mut vec_0 = Vec::with_capacity(usize_0);
5    let i32_0 = 45i32;
6    vec_0.push(i32_0);
7  }
```

```
8
9   // became
10
11  #[test]
12  fn test() {
13    let usize_0 = 32usize;
14    let mut vec_0 = Vec::with_capacity(usize_0);
15    let i32_0 = 45i32;
16    vec_0.push(i32_0);
17    let i32_1 = 106i32;
18    vec_0.push(i32_1);
19    let usize_1 = vec_0.len();
20  }
21
```

Change a statement For a test case $t = (s_1, s_2, \ldots, s_l)$ with length l, each statement s_i is changed with probability $1/n$. If s_i is a primitive statement, then the numeric value represented by s_i is changed by a random value in $\pm[0, \Delta]$ where Δ is a constant. If s_i is not a primitive statement, then either a function or a field with the same return type as $v(s_i)$ and parameters satisfiable with the values in the set $\{v(s_k) \mid 0 \leq k < i\}$ is randomly chosen or the arguments $a_1, \ldots, a_n$ of s_i are changed with the probability $1/n$. Each changed argument a_x is replaced by a random value out of the set $\{v(s_k) \mid 0 \leq k < l\}$ or a generated value. The new value must again satisfy the rules defined in Section 5.1. In Listing 5.2, the first argument (vector instance) of the function call `push` was replaced by another one, which was generated on-the-fly because no other instance of the same type existed in the test before.

Listing 5.2 The first argument (which effectively is the function owner) of the call to *push* in Line 8 has been changed to a newly created value *vec_1*

```
1   #[test]
2   fn test() {
3     let usize_0 = 32;
4     mut vec_0 = Vec::with_capacity(usize_0);
5     let i32_0 = 45;
6     vec_0.push(i32_0);
7     let i32_1 = 106;
8     vec_0.push(i32_1);
9     let usize_1 = vec_0.len();
10  }
11
12  // became
13
14  #[test]
15  fn test() {
```

```
16      let usize_0 = 32;
17      let mut vec_0 = Vec::with_capacity(usize_0);
18      let i32_0 = 45;
19      vec_0.push(i32_0);
20      let i32_1 = 106;
21      let mut vec_1 = Vec::new();
22      vec_1.push(i32_1);
23      let usize_1 = vec_0.len();
24  }
25
```

Delete a statement For a test case $t = (s_1, s_2, \ldots, s_l)$ with length l, each statement s_i can be deleted with probability $1/l$. As the value $v(s_i)$ might be used as an argument in any of the statements $s_{i+1}, \ldots, s_l$, the test needs to be repaired to remain valid. For each statement s_j, $i < j \leq l$, if s_j refers to $v(s_i)$, then this argument is replaced with another value out of the set $\{v(s_k) \mid 0 \leq k < j \wedge k \neq i\}$ which has the same type as $v(s_i)$ and is allowed to be used according to the rules from Section 5.1. If this is not possible, then s_j is recursively deleted as well [44]. In Listing 5.3, an instance of a vector was deleted, affecting the method calls on that instance as well. For the `push` method call, there is no alternative instance of the same type. Since it does not return any value, it can simply be eliminated because no other statements make use of it. The call to the `len` method was transferred to the other instance of the same type present in the test (`vec_1`). If there were no second instance of the same type in the test, or it was already consumed at this point, the statement would have been removed, too.

Listing 5.3 The definition of *vec_0* in Line 6 has been deleted, and statements that used it have been updated

```
1   #[test]
2   fn test() {
3       let usize_0 = 32;
4       let mut vec_0 = Vec::with_capacity(usize_0);
5       let i32_0 = 45;
6       vec_0.push(i32_0);
7       let i32_1 = 106;
8       let mut vec_1 = Vec::new();
9       vec_1.push(i32_1);
10      let usize_1 = vec_0.len();
11  }
12
13  // became
14
15  #[test]
16  fn test() {
```

```
17      let usize_0 = 32;
18      let i32_0 = 45;
19      let i32_1 = 106;
20      let mut vec_1 = Vec::new();
21      vec_1.push(i32_1);
22      let usize_1 = vec_1.len();
23  }
24
```

5.3. Fitness Function

In order to better guide the selection of parents for future generations, all individuals in a population are evaluated according to their fitness. A good fitness function is very important in the search for solutions. Solutions that are better than others in a certain way should be rewarded with better fitness values. Whatever is a better fitness, a higher or lower fitness depends on whether the search strategy tries to maximize or minimize the fitness function [64].

During the search, we employ basic block coverage, i.e., each MIR basic block is a target. When a block is executed, the corresponding test is marked as good with respect to this target. If a function or method is executed but a blokc is not, then the fitness determines the distance of the respective test case to an execution of the target. We adopt the widely used definition of fitness for branch coverage, *approach level* and *branch distance* computation, which is used in search-based generation of test data [64]. The approach level describes how far a test case was from a target in the Control Dependence Graph (CDG) when the test case deviated from the course. We compute approach level in terms of the number of missed control dependencies between the target and the point where the test case took a wrong turn. Approach level is equal to 0 if all control dependency edges were reached by the test case.

The branch distance estimates how far the branch at which execution diverged is from evaluating to the necessary outcome. To determine these values, the test case has to be executed once on the unmodified software. If the target has been executed, then approach level and branch distance are 0. Otherwise, the rules from Table 5.2 are used for calculation of numeric values. For all other cases, the local branch distance is either 0 or 1 depending on whether the respective branch was executed or not. Assume the function `bar` from Listing 5.4 which takes a boolean named `flag`. When called `bar(true)`, the branch distance for the *true* branch is 0 since it gets executed, while the branch distance for the *false* branch is 1 because we do not know how the boolean value has been computed. It could be a result of some numbers comparison, a function call, or just a constant. Figuring that out requires inter-procedural static analysis and advanced MIR instrumentation that RustyUnit currently does not apply.

Listing 5.4 Computation of local branch distance for a boolean flag that we do not know where it came from

```
fn bar(flag: bool) {
    if flag {
      println!("Flag is true");
    } else {
      // ...
    }
}
```

Of course, not every program uses only comparisons of numbers and thus many values for the branch distance will be binary. This means that the approach level might have much more important meaning for the search. Consider the following example function `foo`:

```
fn foo(x: i32, y: i32) -> i32 {
    if x < 5 {
        if y ⩾ 10 {
            0
        } else {
            1
        }
    } else {
        2
    }
}
```

Assume that a test invokes `foo`. In Table 5.1, calls to `foo` with various arguments and the resulting approach level (AL) and branch distance (BD) are shown as examples .

	foo(0, 5)		**foo(6, 1)**		**foo(0, 10)**	
Branch (Line)	AL	BD	AL	BD	AL	BD
3-7	0	0	0	2	0	0
9	0	5	0	0	0	5
4	0	5	1	2	0	0
6	0	0	1	2	0	1

Table 5.1. Fitness value calculation for different invocations of `foo`

As usual in the SBST literature, the branch distance should not dominate the approach level; thus, the former is normalized in the range $[0, 1]$, for instance, using the following normalization function proposed by Arcuri [4]:

$$\alpha(x) = \frac{x}{x+1}$$

Condition	Distance True	Distance False
x == y	|x - y|	1
x != y	1	|x - y|
x >y	y - x + 1	x - y
x >= y	y - x	x - y + 1
x <y	x - y + 1	y - x
x <= y	x - y	y - x + 1

Table 5.2. Local branch distance calculation for numeric values

This allows us to calculate the overall fitness of a test t with respect to target m:

$$D_m(t) = \text{Approach Level} + \alpha(\text{Branch Distance})$$

5.4. Test Oracles and Testability Transformations

A *testability transformation* is a source-to-source program transformation that seeks to improve the performance of some chosen test data generation technique [49], e.g., reach difficult branches. Beyond that, a SUT can be transformed in a way such that artificial branches are inserted into the code to guide the search into triggering some exceptional behavior and crash the program. For instance, EvoSuite [33] employs multiple testability transformations such as array access transformation (Listing 5.5), division by zero transformation, or numerical overflow transformation. Those transformations do not increase code coverage in general but contribute to discovering potential unforeseen behavior and generating inputs that trigger it.

Listing 5.5 Array access transformation in EvoSuite for Java

```
// Original method
void foo(int x) {
  bar[x] = 0;
}

// Transformed method for better testability
void foo(int x) {
  if (x < 0) {
    throw new NegativeArraySizeException();
  }
  if (x ⩾ bar.length) {
    throw new ArrayIndexOutOfBoundsException();
  }
  bar[x] = 0;
}
```

With the new branches in place, artificial coverage objectives are created, which a GA can often compute meaningful distances for, which significantly improves the effectiveness of it in this regard compared to random search. We apply similar ideas to RustyUnit. However, some of the problems that other programming languages have, do not affect (safe) Rust, e.g., there are no null pointers as in C or Java, for which we can weave explicit if conditionals to check nullity. Moreover, the Rust compiler already incorporates checks for common issues in the MIR that are addressed by testability transformations. That is, we do not have to transform the MIR to introduce new branches as in Listing 5.5 but only trace these branches automatically inserted by the compiler.

5.5. Seeding Strategies

According to Fraser's and Arcuri's definition [41], seeding refers to techniques that make use of previous related knowledge to help solve the testing problem given.

A GA usually starts with a random population with constants being chosen randomly, too, in the initial population or during the search. However, domain knowledge of the testing problem given can be exploited to construct the population. RustyUnit applies two seeding strategies that are described in the following to improve the effectiveness and performance of the search.

Seeding Constants Execution flow of complex nested code often depends on specific values, i.e., branch conditions containing constants or string comparisons that perform a constant substring verification. Some search-based tools make use of such data by collecting them from the source code and then seeding the search with the values. RustyUnit also implements this technique and extracts constant values from the MIR of a SUT. Even though utilizing the extracted constants does not necessarily result in covering a branch (e.g., we will not cover the true branch of `if x < 2` with `x = 2`), those values are often very close to the required ones and can be improved by mutation. With the extracted constants in-place, RustyUnit employs them with a certain probability p_{mir} whenever a constant must be generated during the search, for instance, to satisfy a parameter of a function invocation.

Seeding Functions The strategy to generate new test case individuals as described in Algorithm 6 does not guarantee that all of the functions out of the pool M will be incorporated in the generated tests and thus, executed, which is a prerequisite for higher code coverage and better fitness. Therefore, RustyUnit provides the ability to sort the pool of possible functions into a ring buffer

and pick them one-by-one from the buffer each time a new function invocation shall be generated during the initialization of the initial population. In this way, RustyUnit achieves that all practically possible functions are at least entered once in the final test suite. This is ensured by the fact that an function entry is an objective itself, too, so the test that covers it remains in the archive.

6

Implementation

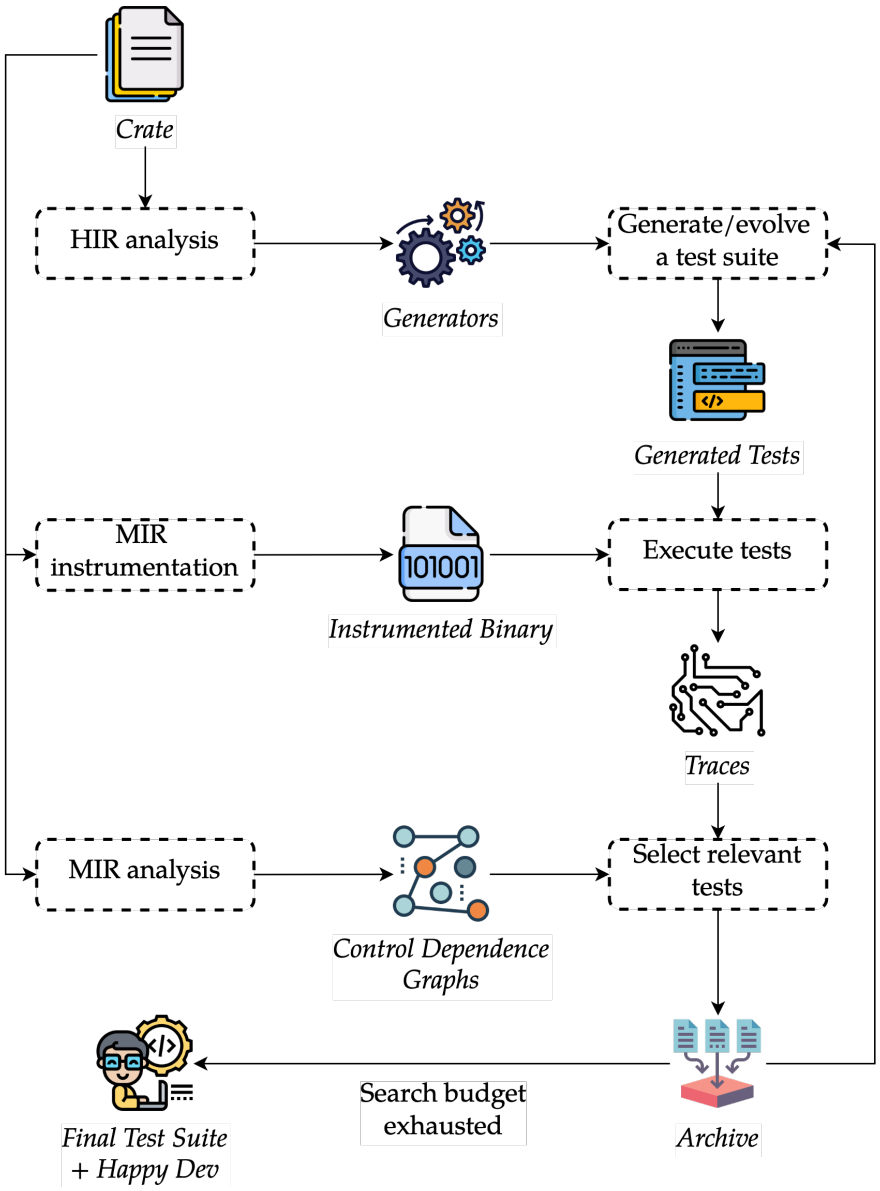
In this chapter we present essential parts of the implementation of RustyUnit. To generate tests for a crate, RustyUnit has to complete several intermediate steps that Figure 6.1 illustrates. First and foremost, the tool requires the information about which data types the *Crate* has and which functions the data types provide in order to be able to model meaningful tests in the first place. Therefore, it performs a HIR analysis, which yields a collection of *Generators*. These provide an overview of which data types can be instantiated in a test case, and which methods can be invoked on those instances. RustyUnit also derives the ownership rules at this point, e.g., whether and how multiple statements may use a certain variable in a valid way, as described in Section 5.1.

To evaluate the *Generated Tests* in terms of their coverage, RustyUnit instruments the MIR and compiles the crate yielding an *Instrumented Binary*. During instrumentation, the tool injects instructions into the MIR to trace the execution of individual basic blocks and branches. If a generated test executes a code location in the crate, the event (and the corresponding branch distance to another branch in case it has been a branch) is stored in the *Execution Traces*.

The execution traces only contain a raw part of the fitness values. To calculate the overall fitness value with respect to each coverage target, RustyUnit must additionally determine the approach level from the corresponding *Control Dependence Graphs*, as described in Section 5.3. It computes the graphs of execution units contained in the crate by analyzing its MIR. This implies building a post-dominator tree from the CFG, which a MIR effectively is, and then computing the control dependencies. With the CDGs and execution traces in place, RustyUnit computes how good a certain test is with respect to the coverage targets, selects the fittest ones, and stores them into the *Archive*. Now, RustyUnit can either generate a new population and evolve it in the next iteration, as already shown in Figure 5.1, or, when the search budget is exhausted, return the archive, i.e., the best test cases found up to that point in the form of Rust source code.

In the next sections, we explain in detail how RustyUnit performs the outlined steps by hooking into the compiler.

Figure 6.1. The architecture of RustyUnit



6.1. Using the Compiler Hooks

The Rust compiler design makes it possible to use it both, as an executable and as a collection of libraries. To this end, Rust provides compiler crates, which can be recognized by their common prefix, `rustc`, such as `rustc_driver`, which is an interface for running the compiler programmatically.

To have access to the compiler internals, multiple things are required. Firstly, the internals are not stable and probably will never be. As such, programs that use compiler internals must use *nightly* versions of the toolchain. Secondly, the crates must have the `rustc_private` feature enabled, which allows crates to access individual compiler crates. Thirdly, these crates must be explicitly specified in the crate root using the `extern crate` keywords. For instance, if the program needs the internal `rustc_driver`, it must specify it the following way in the crate root: `extern crate rustc_driver`.

With compiler internals available, we can implement some of the callbacks that the compiler will call at appropriate point of time. The `rustc_driver::Callbacks` trait defines four callbacks, which are called in the following order:

1. `config`
2. `after_parsing`
3. `after_expansion`
4. `after_analysis`

The `config` method will be called before creating the compiler instance and allows to change the default compiler configuration. Each of the other three functions will be called after the end of the corresponding compilation phase. The interface provides the `after_*` hooks with an instance of the compiler and, more important, with a `Queries` object which allows to query the global context of the crate under compilation. It is very convenient because this way we can extract all the items defined in a crate, including functions along with their parameters and return types.

The sequence of the callbacks is called for each crate and its dependencies since each crate is compiled separately, with the SUT coming last after its dependencies have been processed. This theoretically also provides us the ability to analyze the code structure of the dependencies in case some of the SUT's functions required a parameter of an external type. However, this does not work for the standard library of the language as it usually already comes precompiled. Thus, RustyUnit is not able to extract this data automatically and therefore, would not know how to initialize and manipulate a vector (`std::vec::Vec`), for example. However, since the data is essential for generating tests for most of the crates, RustyUnit includes prepared types from the standard library and their functions relevant to the case study subjects.

Listing 6.1 We can register a listener function to analyze the code structure of the compiled crate

```
fn after_analysis(&mut self, _c: &Compiler,
    _q: &'tcx Queries<'tcx>) {
```

```
  _q.global_ctxt()
    .map(|ctxt| ctxt.peek()
    .enter(analyze))
}

fn analyze(tcx: &TyCtxt<'_>) {
  // Obtain the MIR instance of a function
  let mir = tcx.optimized_mir(
    // Definition ID of the function
  );

  // Extract types and CDGs
}
```

Within the callbacks, with a compiler and a `Queries` object given, we can access the IRs of the crate under compilation, e.g., we can query the MIR of a function body by its `DefId`, as shown in Listing 6.1. To this end, we retrieve the `TyCtxt` instance in Line 5, which is the central data structure of the compiler and provides methods to query different regions of the compiled code, e.g., `optimized_mir` in Line 10 to obtain the MIR of a function. We cannot modify it and pass the modified version back to the compiler, though, which is critical for MIR instrumentation. This is not a concern for the HIR and MIR analysis, as we only want to read the structure of the crate. For this purpose, we can override the default implementation of the `optimized_mir` method with a custom one. The `config` callback provides a mutable `Config` object which allows to replace many of the query functions.

Listing 6.2 The Rust compiler interface accepts an object which implements its callback trait, allowing us to execute code at different compilation phases

```
struct CompilerCallbacks;

type mir_fn_ptr = for<'tcx>
  fn(_: TyCtxt<'tcx>, _: DefId) -> &'tcx Body<'tcx>;

impl rustc_driver::Callbacks for CompilerCallbacks {
  fn config(&mut self, _config: &mut Config) {
    _config.override_queries = Some(|_, _, external| {
      let fn_ptr: mir_fn_ptr = |tcx, def_id| {
        let opt_mir
          = rustc_interface
            ::DEFAULT_EXTERN_QUERY_PROVIDERS
            .borrow().optimized_mir;
        // Query original MIR instance
        let mut body = opt_mir(tcx, def_id).clone();
        let hir_id = tcx.hir()
          .local_def_id_to_hir_id(def.expect_local());
```

```
18
19          if is_monitor(hir_id, &tcx) {
20            // Do not instrument our own monitor
21            return tcx.arena.alloc(body);
22          }
23
24          // Instantiate MIR visitor
25          let mut mir_visitor = MirVisitor { tcx };
26          // Instrument the original MIR
27          mir_visitor.visit_body(&mut body);
28          // Pass instrumented MIR back to compiler
29          tcx.arena.alloc(body)
30        };
31        external.optimized_mir = fn_ptr;
32      });
33    }
34  }
```

As shown in Listing 6.2, a provider for an optimized MIR is a function pointer. Our custom provider implementation (Lines 8-30) is a basic wrapper, which:

1. first obtains the original MIR instance in Line 15 using the default implementation in Lines 10-13,
2. verifies that the current MIR instance is not part of our own monitor in Line 19, otherwise skips the instrumentation,
3. applies the `MirVisitor`, which mutates the MIR in-place, in Line 27,
4. returns the mutated version to the compiler in Line 29.

The compiler invokes the `optimized_mir()` provider in further steps, and thus, propagates the mutated MIR instance.

Listing 6.3 shows how the Rust compiler can subsequently be started programmatically. The program accepts command line arguments intended for the compiler in Line 2, instantiates the struct implementing the aforementioned callbacks in Line 5 and a compiler instance in Line 8, and runs the compiler in Line 10.

Listing 6.3 Running the Rust compiler like a library

```
1  fn run_rustc() -> Result<(), i32> {
2    let rustc_args: Vec<String> = std::env::args()
3      .collect();
4
5    let mut callbacks = CompilerCallbacks {};
6    // Register the callbacks for analysis
7    // and instrumentation and run the compiler
8    let err = rustc_driver::RunCompiler::new(
```

```
    &rustc_args, &mut callbacks
  ).run();

  if err.is_err() {
    return Err(1);
  }

  Ok(())
}

fn main() {
  exit(run_rustc()
      .err()
      .unwrap_or(0)
  )
}
```

Now this program can be used in general for instrumenting Rust programs. However, calling it directly is very cumbersome, since the compiler needs to know many arguments, especially for large crates, in order to link dependencies correctly. Usually one uses a build system for this purpose, e.g., Cargo, which takes over this task and provides appropriate compiler arguments when building a crate. Fortunately, Cargo provides a way to define a custom `rustc` wrapper, which is then supplied with the arguments by Cargo. The task of such a wrapper is to eventually call the real compiler on itself and forward those argument in order to fullfil the expected behavior during compilation [1]. To use the wrapper, one has to set the environment variable `RUSTC_WRAPPER` to the path to the wrapper binary. Finally, a target Rust program can be built and instrumented by running:

```
RUSTC_WRAPPER=./instrumentation cargo +nightly build
```

Since the wrapper uses `rustc_private` features, we have to tell Cargo to enable the nightly channel. The next step is to define how to instrument the MIR so that we can observe the execution paths triggered by an execution of generated tests.

6.2. HIR Analysis

For the purpose of test generation, HIR and MIR are of primary importance. First, we are interested in all possible traits, structs/enums, and their functions and fields in the crate we want to test. This information allows us to evaluate which statements we can use in a generated test at all and what types of instances do those statements produce. We call functions and methods generators since those can generate instances

[1] `http://web.archive.org/web/20220506220810/https://doc.rust-lang.org/cargo/reference/environment-variables.html`

and values of the required types when we recursively search for a dependency. The fact that desugaring has already been performed in the HIR has no influence at this point, because the transformations are only performed intra-procedurally. We are only interested in type and method definitions, their parameters and return types, which still remain unchanged at this level. Moreover, thanks to HIR we can also determine the absolute path of a type, for example a parameter. Relative type names, such as `File` instead of `std::fs::File`, can lead to difficulties when compiling the generated tests due to missing imports. Thus, it is easier to carry a complete type path, and by means of HIR we can easily query this.

The top-level data structure in the HIR is the `Crate`, which stores the contents of the crate currently being compiled. The Rust compiler compiles crates independently and, thus, only ever constructs HIR for the current crate. Technically, the HIR `Crate` structure contains a number of maps that serve to organize the content of the crate for easier access. For instance, the contents of individual items (e.g., modules, functions, traits, impls and so on) in the HIR are not directly accessible in the parents. So, for example, if there is a module item `foo` containing a function `bar()`:

```
1 mod foo() {
2   fn bar() { }
3 }
```

then in the HIR, the representation of module `foo` would only have the ID `I` of `bar()`. To get the details of the function, we would lookup `I` in the `items` map of the HIR, as shown in Listing 6.4. One nice result from this representation is that one can iterate over all items in the crate by iterating over the key-value pairs in these maps (without the need to trawl through the whole HIR). The other reason for such decoupled design is for better integration with incremental compilation. This way, if we gain access to some item, e.g., for the module `foo`, we only gain access to the ID for `bar()`, and we must invoke some functions to lookup the contents of the function given its id. This gives the compiler a chance to observe that we accessed the data for `bar()`, and then record the compilation dependency.

In Listing 6.4 we iterate over the `items` map of the HIR and extract the relevant elements like functions, structs, and methods by pattern matching the type of an item. Each item stores a `Span` object, which shows its origin at the source code level, i.e., the source file it has been defined within. This is important for us because in Rust, unlike, for instance, in Java, unit tests can and should, by convention, be written into the files that contain the code to test. This also means that unit tests, also our generated tests, can invoke private functions and methods directly, which in turn can lead to better search results.

On the other hand, the ability to invoke private items leads to further constraits for our approach. We say that a test case is bound to a module, that is, a source code file, if it calls private functions from that module. When generating further statements for a module-bound test using *GenTest* and *GenObject* from Algorithm 6, RustyUnit is only allowed to select those functions that either belong to the same module as the test at hand or are public to keep the test compilable.

Listing 6.4 Iterate over the items in the HIR of a crate

```
fn hir_analysis(
  tcx: &TyCtxt<'_>, callables: &mut Vec<Callable>
) {
  for item in tcx.hir().items() {
    match &item.kind {
      ItemKind::Fn(sig, _, _) => {
        if &item.ident.name.to_string() != "main" {
          // item is a loose function
          analyze_fn(sig, item.def_id,
            &mut callables, &tcx);
        }
      }
      ItemKind::Impl(impl_item) => {
        // item is an associative function
        analyze_impl(impl_item, &mut callables, &tcx);
      }
      ItemKind::Struct(s, g) => {
        // item is a struct
        analyze_struct(item.def_id, s, g, &item.vis,
          file_path.unwrap(), &mut callables, &tcx);
      }
      // ... enums, traits
    }
  }
}
```

With the HIR analysis set up, RustyUnit can already generate random tests since we know which functions can be called. Listing 6.5 shows an example program to test. After the analysis, the following holds in terms of Algorithm 7:

$$M = \left\{ \begin{array}{l} \texttt{Address::new} \rightarrow \texttt{Address} \\ \texttt{Person::new} \rightarrow \texttt{Person} \\ \texttt{Person::address} \rightarrow \texttt{\&Address} \\ \texttt{say\_hello\_to} \rightarrow \texttt{()} \\ \texttt{address\_to\_string} \rightarrow \texttt{String} \end{array} \right\}$$

If we want to execute the `address_to_string` function in an empty test $t = \{\}$, we have to generate an argument of type `Address`. Thanks

to the information about the return type of a method, we can use $GenObject(Address, \{\}, M, t)$ to create either an `Address` object using `Address::new` or `Person::address`. The latter one expects a `self` argument which would require us to recursively search for a way to create an object of type `Person`, e.g., by calling the constructor `Person::new`.

Listing 6.5 After HIR analysis, we know how a `Person` object can be generated to be used in `say_hello_to`.

```
struct Address {
  street: String,
  house_n: usize,
  city: String,
  zip: usize
}

impl Address {
  pub fn new(street: &str, house_n: usize,
  city: &str, zip: usize) -> Self {
    Address {
      street: street.to_owned(),
      house_n,
      city: city.to_owned(),
      zip
    }
  }
}

struct Person {
  name: String,
  address: Address
}

impl Person {
  pub fn new(name: &str, address: Address) -> Self {
    Person {
      name: name.to_owned(),
      address
    }
  }

  pub fn address(&self) -> &Address {
    &self.address
  }
}

fn say_hello_to(person: &Person) {
```

```
39      // ...
40  }
41
42  fn address_to_string(address: &Address) -> String {
43      // ...
44  }
```

6.3. MIR Analysis

In this section, we describe the analysis steps RustyUnit performs on the MIR of a crate under test.

6.3.1. Control Dependence Graphs

To improve the effectivity of generated tests, both DynaMOSA and the fitness function (approach level) that RustyUnit uses as described in Section 5.3, exploit information from the CDGs [31] of the SUT. A CDG can be constructed from a CFG and its post-dominator tree. As MIR itself is a CFG, we can easily build the post-dominator-tree and the CDG of each method and function we want to test. RustyUnit does not use the complete CFG of a `Body`, but rather a subset of it. The Rust compiler creates many unwind edges in the MIR that point to basic blocks whose task is to cleanup the heap in case the program panics at some point of its execution. However, we are not interested in the cleanup logic of the compiler. Moreover, considering unwind nodes results in CFG having multiple exit nodes, which complicate the computation of a post-dominator tree. We filter out most of the unwind edges from a CFG except for those described in Section 6.4.3. To simplify further tree computations, we add a dummy exit node that we point all original exit nodes to, in case if there are at least two dedicated exit nodes.

The standard algorithm to compute a dominator tree from a CFG yields a post-dominator tree when applied to reversed CFG, given that the original CFG has a single exit point, i.e., a single entry point with reversed edges.

Prosser [73] introduced the notion of dominance in a 1959 paper on the analysis of flow diagrams, defining it as follows: Box i dominates box j if every path (leading from input to output through the diagram) which passes through box$j$ must also pass through box i. Thus box i dominates box j if box j is subordinate to box i in the program. He did not specify the algorithm to compute dominance, though. Since then, several algorithms for calculating dominance have been presented. Post-dominance can be computed the same way on a CFG with reversed edges, given that the CFG has a single exit point. RustyUnit exploits the `petgraph` graph library for Rust, which provides an implementation of the simple fast dominance algorithm [25] by Cooper et al.

Finally, we compute a CDG for each method body based on the algorithm described by Ferrante et al [31]. At the end of this step, RustyUnit yields a set of CDGs, one for each function in the SUT.

6.3.2. *Constant Pool Analysis*

As part of the seeding strategy for enhanced search described in Section 5.5, RustyUnit additionally performs an analysis of the constants used in the SUT to simplify the coverage of complex branches. Assume an example program that parses raw date components to a `Date` instance:

```
struct Date;

impl Date {
  fn from(day: u8, month: &str, year: u16)
      -> Result<Date, Error> {
    if month.to_lowercase().contains("feb")
        && day == 29 {
      // do leap year things ...
    }

    // ...
  }
}
```

The associated function `from` checks at the beginning whether a leap year can be inferred from the given arguments. For the execution of the program to enter the true branch, a test has to call the function with 29 for the `day` parameter and a string containing "month" for the `month` parameter. The probablity that a test will hit this branch with a combination of randomly generated values is quite low. Altough for `day` only 8-bit integer numbers can be used with a value range of $[0, 255]$, it gets much more complicated when we try to generate an approapriate string value randomly, depending on the parameters for the maximum length and allowed symbols.

However, by analzing the relevant snippet of the MIR, we can learn what constant values the given function uses.

```
fn Date::from(_1: u8, _2: &str, _3: u16)
    -> Result<Date, Error> {
  // Locals definition

  bb0: {
    _6 = _2;
    _5 = <impl str>::contains::<&str>(move _6,
        const "feb")
      -> bb4;
```

```
10    }
11
12    bb1: {
13      _4 = const false;
14      goto -> bb3;
15    }
16
17    bb2: {
18      _8 = _1;
19      _7 = Eq(move _8, const 29_u8);
20      _4 = move _7;
21      goto -> bb3;
22    }
23
24    bb3: {
25      SwitchInt(move _4) -> [false: bb9, otherwise: bb5];
26    }
27
28    bb4: {
29      SwitchInt(move _5) -> [false: bb1, otherwise: bb2];
30    }
31
32    // Other blocks
33  }
```

The shown snippet is typical pattern which the Rust compiler creates to incorporate short-circuit evaluation of groupped conditionals. The first block check if the input string contains the substring "feb" and stores the result of the comparison into _5 in Line 7. Then, block 4 is executed, which decides on the basis of the result whether the second part of the coniditional, i.e., block 2, shall be executed or not. If yes, its result is stored in _4 in Line 19, otherwise _4 is assigned a constant false value. In any case, the value of _4 is finally read in block 3 and either the branch for the leap year is entered (block 5) or skipped.

As in the textual representation of the MIR shown, constant values have the prefix `const` and under the hood, they are of type `Constant`, which RustyUnit looks for when visiting the MIR data structure and stores their literal value and type. At the end of the analysis, RustyUnit reports the set of constants used in the SUT.

6.4. MIR Instrumentation

Tests that are generated by a search-based technique need to be executed at least once to provide some feedback on their fitness in regard to the SUT. We only can know what parts of a SUT a test reaches if we execute it and inspect the execution flow. At this point, RustyUnit collects the coverage and distance data, i.e., local fitness values.

The only compiler callback we implement to this end is the `config` function, which allows us to modify the MIR of a target program on-the-fly. The internal compiler crates already provide a `MutVisitor` [2] that can be used to traverse and modify MIR objects. To trace the execution of branches of a SUT, RustyUnit inserts additional instructions at relevant places in the MIR. Those are mainly entry basic blocks (to trace if a body has been entered at all during an execution of a test) and basic blocks that introduce branches and split the execution flow, e.g., those having `SwitchInt` and `Assert` terminators. Technically, there are also other terminators which have more than one successor, e.g., `Calls`, which are function invocations. However, the optional additional branch is used to point to basic blocks which unwind allocated memory in case the function being called panics, i.e., the program crashes. However, those blocks are compiler-generated and not interesting; thus, RustyUnit only analyzes and instruments blocks until the beginning of an unwinding chain, then stops and continues with others. Listings 6.6 and 6.7 show an example function to instrument and the relevant parts of its MIR.

Listing 6.6 Example function to instrument

```
fn foo(x: i32, y: i32) {
  if x < y {
    println!("x < y");
  } else {
    println!("x ≥ y");
  }
}
```

Within the entry basic block in the MIR, the two input arguments are compared and their result is stored into the local variable `_3`. Then, based on the comparison result, the execution of the function can either jump to the block `bb1` (if `x` was less than `y`) or `bb3` (if `x` was greater or equal to `y`), that is, there are two branches at this point.

Listing 6.7 MIR of the `foo` function

```
fn foo(_1: i32, _2: i32) -> () {
  // Locals definition

  bb0: {
    _4 = _1;
    _5 = _2;
    _3 = Lt(move _4, move _5);
    SwitchInt(move _3) -> [
      false: bb2, otherwise: bb1
    ];
```

[2] http://web.archive.org/web/20220506220855/https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/visit/trait.MutVisitor.html

```
  }

  bb2: {
    // print "x ⩾ y"
  }

  bb1: {
    // print "x < y"
  }

  // Other blocks
}
```

For simplicity, we will consider individual instrumentation methods separately in the next chapters. Effectively, however, they will all be used together to trace various aspects of program execution.

6.4.1. Body Entry Instrumentation

To trace the entry of a function, we have to insert a call to our monitor at the very beginning. A function call is always a terminator in MIR; that is, we have to insert a whole new basic block for each trace function invocation. It must be the very first block in the sequence of basic blocks of the respective function, i.e., `bb0`, so we shift the original blocks by one and let our new artificial tracing block point to the original entry block, which is `bb1` now. Since the MIR graph of each function is an independent data structure, they all have independent identificators, e.g., the ID of the module and the function currently being compiled. The combination of the two allows us to uniquely identify function bodies during execution. The IDs are assigned by the compiler and are already known at the time of instrumentation, which is why RustyUnit only needs to weave them in as constants. Assume that the ID of the module in which the function `foo` is defined is equal to 2, while the ID of the function itself is 3. Then, after instrumenting the entry of the function, its MIR looks like:

```
fn foo(_1: i32, _2: i32) -> () {
  // Locals definition

  bb0: {
    _3 = monitor::trace_entry(
      const 2_u64,
      const 3_u64
    ) -> bb3;
  }

  bb1: {
    _4 = _1;
```

```
13      _5 = _2;
14      _3 = Lt(move _4, move _5);
15      SwitchInt(move _3) -> [false: bb3, otherwise: bb2];
16    }
17
18    bb3: {
19      // print "x ⩾ y"
20    }
21
22    bb2: {
23      // print "x < y"
24    }
25
26    // Other blocks
27  }
```

The entry block is now a call to our `monitor` in Line 5 that traces the entry of the function. It points to the original entry block, which is now `bb1` (Line 11) since RustyUnit shifted all other blocks by one in the blocks array of the function body. Figure 6.2 illustrates the CFG of the function before and after the instrumentation.

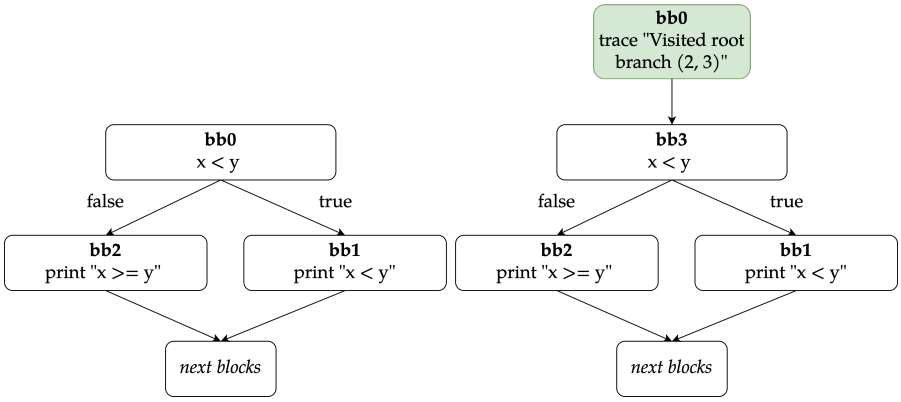
The monitor, which is called to trace the artificial entry block, is a wrapper around the branch distance and tracing logic. Rust does not allow global static and non-constant instances of non-primitive types. That is, we need to log the data outside of the SUT's process, e.g., by writing to a file. By outsourcing calculations and the actual tracing to external `monitor::trace_*` functions of our monitor, we need to expend minimal effort to weave in tracing code, e.g., opening a connection to a traces server, at this low level and can replace it easily by another log functionality. The `monitor` module itself is a regular Rust source file that RustyUnit copies into a crate before compiling and instrumenting it. With this approach, `monitor` is compiled with the actual crate and is available to the instrumented code, so we can call its functions directly. Assume that the `main` function of a SUT looks like this:

```
1  fn main() {
2    foo(10, 20);
3    foo(10, 20);
4  }
```

Then, we will get the following trace output after the instrumentation of the program:

```
1  Entered branch (2, 3)
2  Entered branch (2, 3)
```

Figure 6.2. Instrumentation of a function's entry point



6.4.2. *Branch Instrumentation*

To trace the execution of a decision branch, we need to insert calls to the monitor after a branch has been taken and just before it starts to execute. Doing so, we first update a terminator where branching begins, e.g., a `SwitchInt`, to point to our artificial tracing blocks. Here, the number of tracing blocks per branch is equal to the number of branches. This is due to the fact that after a particular branch is taken, we want to trace the distance for all branches. This means that for each possible branch a call to the monitor is needed, i.e., one basic block. At this point, we know on what basis a particular branch is taken and can thus incorporate appropriate logic.

To this end, RustyUnit analyzes basic blocks backwards to find the definition of a condition operand as described in Algorithm 8. First, the tool checks whether the given local is an argument of the function. If it is, the algorithm stops. Otherwise, it iteratively processes statements of the given block that may assign the local. If the right hand side of such assignment is a constant or a result of a function call, then it is returned since we cannot further specify the origin of v or do any interprocedural analysis. If it is a result of an operation, e.g., $a > b$, the algorithm returns the type of the operation and the operand(s) which we can calculate a distance from. Otherwise, it is an assignment from another variable v', and we try to determine the definition of v'. In case none of the statements in the given block define the local v, the algorithm switches to the preceding blocks and so on.

We insert tracing instructions based on the definition type of the local used in a conditional. For instance, if it is a boolean function argument, then we can only trace 0 and 1 as distances for the true and false branches. On the other hand, the conditional operand is a result of a local operation, then we can compute the exact branch distance based on that operation, as we do in Listing 6.8 for the `foo` function

Algorithm 8 FindDefinition(v, bb)

Input
  v Local to find definition of
  bb Basic block to start from
Output
  d Definition of the local
if v is function argument **then**
  return v
end if
for $s \in$ statements in block **do**
  if s assigns v' to v **then**
    if v' is constant or invocation return value **then**
      return v'
    else
      if v' is math operation **then**
        return Op and operand(s)
      else
        return FindDefinition(v', bb)
      end if
    end if
  end if
end for
$P \leftarrow$ predecessors of bb
for $p \in P$ **do**
  $v' \leftarrow$ FindDefinition(v, p)
  if v' is not empty **then**
    return v'
  end if
end for

defined in Listing 6.7. In the end, we put a chain of tracing blocks with length $l >= 2$ into each branch. In Listing 6.8, `foo` has two branches; thus, we insert the two tracing chains (`bb4`, `bb5`) and (`bb6`, `bb7`). The last block of each tracing chain must point to the original target block of the branching terminator to maintain the original execution flow of the SUT.

Listing 6.8 Instrumented branches in the MIR of the `foo` function

```
fn foo(_1: i32, _2: i32) -> () {
  // Locals definition
  // ...
  let mut _11: monitor::BinaryOp;
  let mut _26: monitor::BinaryOp;
  let mut _34: monitor::BinaryOp;
  let mut _40: monitor::BinaryOp;
```

```
8
9
10  bb0: {
11    _4 = _1;
12    _5 = _2;
13    _3 = Lt(move _4, move _5);
14    SwitchInt(move _3) -> [false: bb4, otherwise: bb6];
15  }
16
17  bb2: {
18    // print "x ⩾ y"
19  }
20
21  bb1: {
22    // print "x < y"
23  }
24
25  bb4: {
26    _8 = _1;
27    _7 = move _8 as f64;
28    _10 = _2;
29    _9 = move _10 as f64;
30    discriminant(_11) = 1;
31    _6 = monitor::trace_branch_bool(const 2_u64,
32                        const 3_u64,
33                        const 2_u64,
34                        move _7,
35                        move _9,
36                        move _11,
37                        const false) -> bb5;
38  }
39
40  bb5: {
41    _23 = _1;
42    _22 = move _23 as f64;
43    _25 = _2;
44    _24 = move _25 as f64;
45    discriminant(_26) = 1;
46    _21 = monitor::trace_branch_bool(const 2_u64,
47                        const 3_u64,
48                        const 1_u64,
49                        move _22,
50                        move _24,
51                        move _26,
52                        const true) -> bb2;
53  }
```

```
54
55    bb6: {
56      _31 = _1;
57      _30 = move _31 as f64;
58      _33 = _2;
59      _32 = move _33 as f64;
60      discriminant(_34) = 1;
61      _29 = monitor::trace_branch_bool(const 2_u64,
62                           const 3_u64,
63                           const 2_u64,
64                           move _30,
65                           move _32,
66                           move _34,
67                           const false) -> bb7;
68    }
69
70    bb7: {
71      _37 = _1;
72      _36 = move _37 as f64;
73      _39 = _2;
74      _38 = move _39 as f64;
75      discriminant(_40) = 1;
76      _35 = monitor::trace_branch_bool(const 2_u64,
77                           const 3_u64,
78                           const 1_u64,
79                           move _36,
80                           move _38,
81                           move _40,
82                           const true) -> bb1;
83    }
84
85    // Other blocks
86  }
```

In tracing chains (`bb4`, `bb5`) and (`bb6`, `bb7`) we want to invoke appropriate monitor functions and store all the information we need to identify and compute the branch distance. Figure 6.3 demonstrates the CFG of the function before and after instrumentation of its branches. As with `monitor::trace_root` in Section 6.4.1, for each tracing call, we specify the IDs of the module and function, as well as the ID of the branch which the tracing runs within. The branch identificator is usually the number of the block the branching terminator originally pointed to, e.g., in this case we have to branch IDs: 1 and 2. We also store the the operands of the binary operation in the `if` conditional, as well as the binary operation itself. In the end, the polarization of the branches remains, i.e., `true` or `false`. With this information we can distinguish in the monitor whether the given branch was hit or not and

determine the distance using the values and the binary operation reported. For example, assume that we invoke `foo(10, 20)` again; then, we get the following trace output:

```
// bb6 gets executed
Distance to branch (2, 3, 2) is 10.0

// bb7 gets executed
Distance to branch (2, 3, 1) is 0.0
```

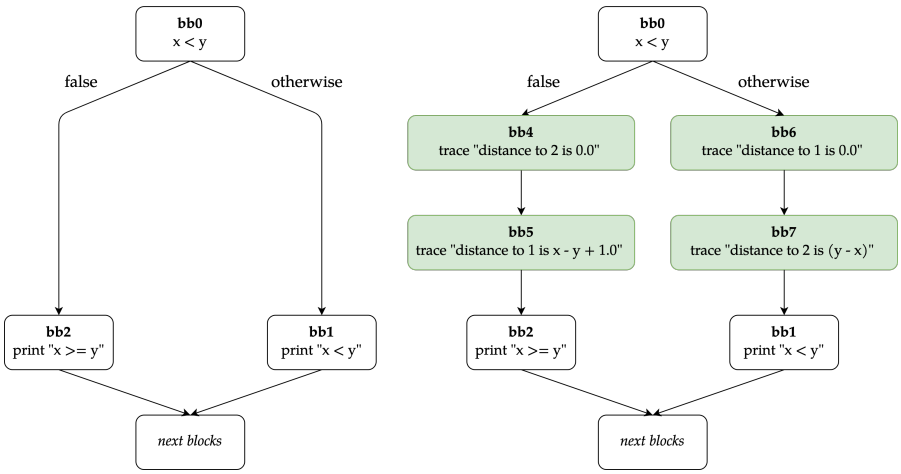
Since $10 < 20$, the execution takes the `true` branch, i.e., the trace chain of blocks `bb6` and `bb7` are run. In `bb6`, we trace the `false` branch while the actually executed branch is `true`. We know that because we can also evaluate the operation `monitor::BinaryOp::LT` with the operands 10 and 20 in the monitor. To calculate the local branch distance, we resort to the formulas in Table 5.2.

For reference, Listing 6.9 shows how the basic block `bb4` would look like if it was written in regular Rust.

Listing 6.9 How would `bb4` look like in regular Rust code

```
monitor::trace_branch_bool(
  2u64, // module ID
  3u64, // function ID within the module
  3u64, // branch block ID within the function
  x as f64,
  y as f64,
  monitor::BinaryOp::LT,
  false // branch polarization
);
```

Figure 6.3. Instrumentation of regular branches



6.4.3. *Testability Transformation Instrumentation*

In the following, we describe several transformations implemented in RustyUnit, some of which were presented by Fraser and Arcuri [33] for the EvoSuite tool. The transformations are implemented at the MIR level. To do so, we need to introduce additional tracing blocks at crucial points. The Rust compiler transforms input programs in the same way as it is done with testability transformations, which are described in Section 5.4. It inserts assertions at relevant places to check for unexpected behavior. For instance, accessing an array with an inappropriate index results in a program crash. This happens because the compiler inserts checks before each array access that shall verify whether a given index is within the bounds of the array. If not, the program panics. Assume the following function that takes an index and an `i32`-array of size 3 and returns the value at the given position:

```
fn get(pos: usize, array: &[i32; 3]) -> i32 {
  array[pos]
}
```

Listing 6.10 shows the relevant part of the function's MIR. Right before the block that accesses the array (Line 16), the compiler inserted an assertion in Line 8 that checks whether the index is less than the length of the array, which is a static constant. If the index is greater than or equal to the length, the program crashes. The other case, i.e., negative index, cannot occur since its value must always be of type `usize`, i.e., an unsigned type, which is guaranteed by the compiler again. Thus, there is no reason to verify this case.

Listing 6.10 The compiler inserts runtime assertions at places where arrays are accessed by index

```
1  fn get(_1: usize, _2: &[i32; 3]) -> i32 {
2    // Locals  definition
3
4    bb0: {
5      _3 = _1;
6      _4 = const 3_usize;
7      _5 = Lt(_3, _4);
8      assert(move _5,
9          "index out of bounds:
10           the length is {} but the index is {}",
11         move _4,
12         _3) -> bb1;
13   }
14
15   bb1: {
16     _0 = (*_2)[_3];
17     return;
```

```
    }
}
```

That is, the compiler inserts an assertion that checks whether the index is less than the length of the array, which is statically constant. If the index is greater than or equal to the length, the program crashes. The other case, i.e., negative index, cannot occur since the index must always be of type `usize`, an unsigned integer type. Thus, there is no reason to verify this case.

Now, we need to insert tracing blocks for both, the "good" and the "bad" case. We could insert two branches always before an assertion block. However, in this case we would have to insert another branching block s, for example with `SwitchInt`, and if applicable we would have to change the original preceding assertion block so that it points to s. In the block s we would also have to copy the logic from the assertion block, which makes the whole approach even more complicated. Another option would be to change the assertion itself. An assertion in MIR has an optional unwind edge which, in the "bad" case, points to a sequence of blocks to clean up before the program crashes, if there is anything to clean up. Assume the function `get` from Section 6.4.3 does some allocations before accessing the array:

```
fn get(pos: usize, array: &[i32; 3]) -> i32 {
  let s = String::from("I'm using heap");
  array[pos]
}
```

Then, the the MIR of the function becomes a bit different:

```
fn get(_1: usize, _2: &[i32; 3]) -> i32 {
  // Locals  definition

  bb0: {
    _3 = <String as From<&str>>::from(
      const "I'm using heap"
    ) -> bb1;
  }

  bb1: {
    _4 = _1;
    _5 = const 3_usize;
    _6 = Lt(_4, _5);
    assert(move _6,
      "index out of bounds:
        the length is {} but the index is {}",
      move _5,
      _4
    ) -> [success: bb2, unwind: bb4];
  }
```

```
21
22    bb2: {
23      _0 = (*_2)[_4];
24      // Drop the string
25      drop(_3) -> bb3;
26    }
27
28    bb3: {
29        return;
30    }
31
32    bb4 (cleanup): {
33      // Drop the string
34      drop(_3) -> bb5;
35    }
36
37    // Other blocks
38  }
```

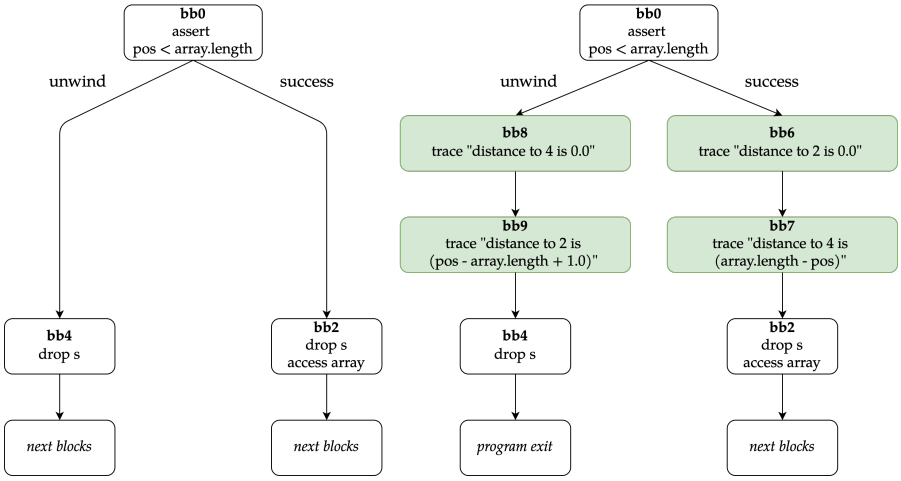
The assertion has now got a second edge. Regardless of whether the index is valid or not, the string will be cleaned up in any case. The behavior is similar to the `finally` block from the `try-catch` construct in Java. We exploit the unwind edge of assertions: if an assertion has an unwind edge, RustyUnit inserts the tracing chain between the assertion and and the existing cleanup blocks. Otherwise, we create an unwind edge manually and connect it to the tracing chain. The instrumentation of assertions is illustrated in Figure 6.4. Since there are two branches, RustyUnit adds a tracing chain of two tracing blocks to each branch. So in case the index is not valid, we the executed path is traced as usual before the proram exits. Thus, we know that the respective test has covered the path. In addition, we get a guidance for the two possible cases.

We do the same for the case of overflows and underflows. Listing 6.11 demonstrates an example of the `add` function that performs addition of two eight-bit integers and the relevant basic block from its MIR. In its MIR, the compiler inserts an assertion about the value of the result to catch a possible overflow. RustyUnit instruments the assertion and insert our tracing chains into the body of the function as described above.

Listing 6.11 MIR overflow check created by the compiler

```
1  fn add(a: i8, b: i8) -> i8 {
2    a + b
3  }
4
5  bb0: {
6    _3 = _1;
```

Figure 6.4. Instrumented assertion




```
7     _4 = _2;
8     _5 = CheckedAdd(_3, _4);
9     assert(
10      !move (_5.1: bool),
11      "attempt to compute `{} + {}`,
12        which would overflow",
13      move _3,
14      move _4
15    ) -> bb1;
16  }
```

6.5. Handling Generics and Traits

A large part of SBST approaches is applied to managed languages such as Java, since these offer a rich set of tools and features like reflection. Reflection allows to analyze the code of the SUT and extract information needed for generating tests by querying the appropriate language API, like `getGenericParameters` of a `java.lang.reflect.Method` instance to get generic parameters. However, when tests are generated for generic elements, it is sometimes difficult to replace a generic type parameter with a matching concrete type because execution depends on the one concrete type [35]. As shown in Listing 6.12, an execution path for generic methods can be affected by checks for the concrete type of a generic instance. Many approaches to analyzing Java programs operate at the bytecode level after type erasure has already happened. This means that bytecode does not know anything about generics, they are replaced, for example, by `Object`.

Listing 6.12 The execution path of the generic Java method depends on the concrete type of the argument

```
public class GenericsExample<T, K> {
  public int typedInput(T value) {
    if (value isinstanceof String) {
      return 0;
    } else if (value isinstanceof Integer) {
      return 1;
    } else if (value isinstanceof Double) {
      return 2;
    } else {
      return 3;
    }
  }
}
```

Similar approach is also possible in Rust to find out the concrete type of a generic, as shown in Listing 6.13. A crucial requirement of this is the conversion of the generic type to `dyn Any`. Then, we can check the runtime type of `value`. The downcasts are also part of MIR and thus can theoretically be used to determine possible suitable types for a generic to narrow down the space of possible types. However, since generics with trait bounds are often a more suitable option, this feature is quite rare and cannot be found in the evaluation subjects, which makes us rely on a more general approach.

Listing 6.13 The execution path of the generic Rust function depends on the concrete type of the argument

```
use std::fmt::Debug;
use std::any::Any;

fn typed_input<T: Any>(value: &T) -> i32 {
  let value_any = value as &dyn Any;

  if let Some(as_string)
      = value_any.downcast_ref::<String>() {
    return 0;
  } else if let Some(as_int)
      = value_any.downcast_ref::<i32>() {
    return 1;
  } else if let Some(as_double)
      = value_any.downcast_ref::<f64>() {
    return 2;
  } else {
    return 3;
  }
}
```

Algorithm 9 GetNecessaryBindings(G, C, T)

Input
  G A generic type
  C A concrete type
  T Type binding
if G is a type parameter **then**
  $T[G] \leftarrow C$
else
  if G is a reference and C is a reference **then**
    $G_r \leftarrow$ referenced type of G
    $C_r \leftarrow$ referenced type of C
    $GetNecessaryBindings(G_r, C_r, T)$
  else
    $I_g \leftarrow$ inner generics of G
    $I_c \leftarrow$ inner generics of C
    $n \leftarrow |I_g|$
    for $i \in (1, n)$ **do**
      $GetNecessaryBindings(I_g[i], I_c[i], T)$
    end for
  end if
end if

Whenever RustyUnit needs to bind a generic type parameter with an actual one, it applies Algorithm 9. The algorithm returns a type binding, which is a mapping from generic to actual types. Assume that we want to know how the type parameters T and E should be replaced in `Result<Box<T>, E>` to get a `Result<Box<MyType>, MyError>`. Algorithm 9 starts from the most outer type, i.e., `Result`. Since the given type is not a type parameter and also not a reference, the algorithm unwraps the generics of the outer type and recursively tries to map `Box<T>` to `Box<MyType>` and `E` to `Error`. `E` is a type parameter, so we store the binding `{E → Error}`. `Box` is a not a type parameter and not a reference either. In the last step, the algorithm stores the additional binding `{T → MyType}`.

The problem of generics becomes relevant whenever the test generation algorithm attempts to instantiate a new instance of a generic type, or to satisfy a parameter for a newly inserted method call. Assume that our test generation algorithm decides to add a call to the method `get(self: &Foo<T>)` of the struct `Foo<T>`, which is defined and implemented as follows:

```
1 struct Foo<T> {
2   bar: T
3 }
4
5 impl<T> for Foo<T> {
```

```
  fn new(bar: T) -> Self {
    Self { bar }
  }

  fn get(&self) -> Option<T> {
    // ...
  }

  fn update(&mut self, bar: T) {
    // ...
  }
}
```

From the HIR analysis we know that `T` is a type parameter of the struct and is not bound by any other constraints. This means that RustyUnit can freely replace `T` by any type in that case. To keep the testing effort smaller, we always check first whether we can fill a generic type parameter by a primitive type from the standard library, if the constraints allow this. Here, for instance, the algorithm could choose `isize` as a good candidate.

The strategy is to first determine the return type of the selected method and then create the potentially necessary dependencies backwards based on that. That is, we invoke `Foo::get(&self)` and store the return value into a variable of type `Option<isize>`. We obtain a type binding *TB* with `GetNecessaryBindings(Option<T>, Option<isize>, TB)`. The binding {`T` → `isize`} between the generic type parameter and the concrete type is stored for the variable for later use.

```
#[test]
fn rusty_test_0() {
  let mut option_0: Option<isize> = Foo::get(&foo_0);
}
```

The test is not compilable as is, though. This is why we use `GenObject` to create an instance of `Foo`, e.g., by invoking its constructor. The constructor has a parameter of type `T` which we already have a binding for; it effectively makes us able to call the constructor with a random integer:

```
#[test]
fn rusty_test_0() {
  let mut isize_0 = 42isize;
  let mut foo_0: Foo<isize> = Foo::new(isize_0);
  let mut option_0: Option<isize> = Foo::get(&foo_0);
}
```

Generics play a very important role in Rust. It is not a typical object-oriented language and instead of using direct types, for example as

method parameters, as is often the case in Java, generics are rather used in conjunction with trait bounds. Trait bounds define what behavior a particular generic type must be capable of. This reduces the number of possible data types a type parameter can be filled with. Listing 6.14 implements the `area` function in Rust, which takes a parameter of generic type `T`. The type parameter `T` must implement the `HasArea` trait, which allows `area` to access the trait's functions.

Listing 6.14 A function that takes a generic types and specifies a bound

```
trait HasArea {
  fn area(&self) -> f64;
}

struct Rectangle { length: f64, height: f64 }

impl HasArea for Rectangle {
  fn area(&self) -> f64 { self.length * self.height }
}

// `T` must implement `HasArea`. Any type which meets
// the bound can access `HasArea`'s function `area`.
fn area<T: HasArea>(t: &T) -> f64 { t.area() }
```

Types that implement a particular trait must thus be found statically during compilation of the SUT and annotated. The compiler provides APIs to query all traits and the collection of the respective `impl` blocks, not only in the compiled SUT, but also in its dependencies and in the standard library itself, as shown in Listing 6.15. Each `impl` block associates with a specific type, i.e., a struct or enum. Thus, we can transitively learn if a type implements a certain trait.

Listing 6.15 Query types for a trait

```
fn get_types_for_trait(trait_id: DefId,
    tcx: &TyCtxt<'_>) -> Vec<Type> {
  let trait_impls = tcx.trait_impls_of(trait_id);
  let types = trait_impls.iter()
    .map(|im| tcx.type_of(im))
    .collect::<Vec<_>>();

  // substitue blanket implementations
}
```

Things get more complicated when a trait is generic because it can be implemented multiple times for the same type with different values of the type parameters. Listing 6.16 demonstrates an example implementation of the `std::convert::From` trait for the type `MyInt` with a 32-bit integer and a string. Due to the increasing complexity of handling generic traits, RustyUnit can only analyze simple cases, i.e., non-

generic traits and traits with no recursive generics, while ignoring the rest.

Listing 6.16 Example implementation of the `std::convert::From` trait from the Rust standard library

```
pub trait From<T> {
  fn from(T) -> Self;
}

struct MyInt {
  value: i32
}

impl From<i32> for MyInt {
  fn from(x: i32) -> MyInt {
    MyInt { value: x }
  }
}

impl From<String> for MyInt {
  fn from(s: String) -> MyInt {
    let value = s.parse::<i32>().unwrap();
    MyInt { value }
  }
}
```

In general, RustyUnit mainly uses primitive data types for a SUT to keep the generated tests simple as far as this satisfies the defined trait bounds. To create a call to the example function `FooBar::foo` (Listing 6.17) in a generated test, we do the following:

Listing 6.17 A generic type A is used in multiple parameters and return value

```
struct FooBar<A> {
  // ...
}

impl<A> for FooBar<A> {
  fn new() -> Self {
    // ...
  }

  fn foo(&self, x: A, v: &Vec<A>) -> Option<A> {
    // ...
  }
}
```

1. Look for a type that satisfies `A`; `A` has no bounds, so bind it to a primitive type, e.g., `i32`. Now, with that information, we

know that the parameters `&self`, `x`, and `v` have to be of type `FooBar<i32>`, `i32`, and `Vec<i32>`, respectively.

2. Using `GenObject` we find that an instance of `FooBar<A>` can be created by the constructor `FooBar::new()`, thus, we define `let f: FooBar<i32> = FooBar::new();`.

3. Generating a primitive like `i32` is just a matter of defining some random value, like `let x: i32 = 42i32;`.

4. Using `GenObject`, find a generator that returns a `Vec<i32>` or a `Vec<T>` with some generic type `T`.

5. From our custom type definition, we know that `Vec::new()` returns a `Vec<T>`. We need its inner elements to be of type `i32`, which does satisfy the generic type `T`, so we take it and define `let v: Vec<i32> = Vec::new();`.

6. Now, we can use the defined variables to generate a call to the `FooBar::foo` and store its returning value, whose bound type we are aware of, too, as shown in Listing 6.18.

Listing 6.18 An example test that invokes `FooBar::foo`

```
#[test]
fn rusty_test_0() {
  let foobar_0: FooBar<i32> = FooBar::new();
  let i32_0: i32 = 42i32;
  let vec_0: Vec<i32> = Vec::new();
  let option_0: Option<i32> = FooBar::foo(
    &foobar_0, i32_0, &vec_0
  );
}
```

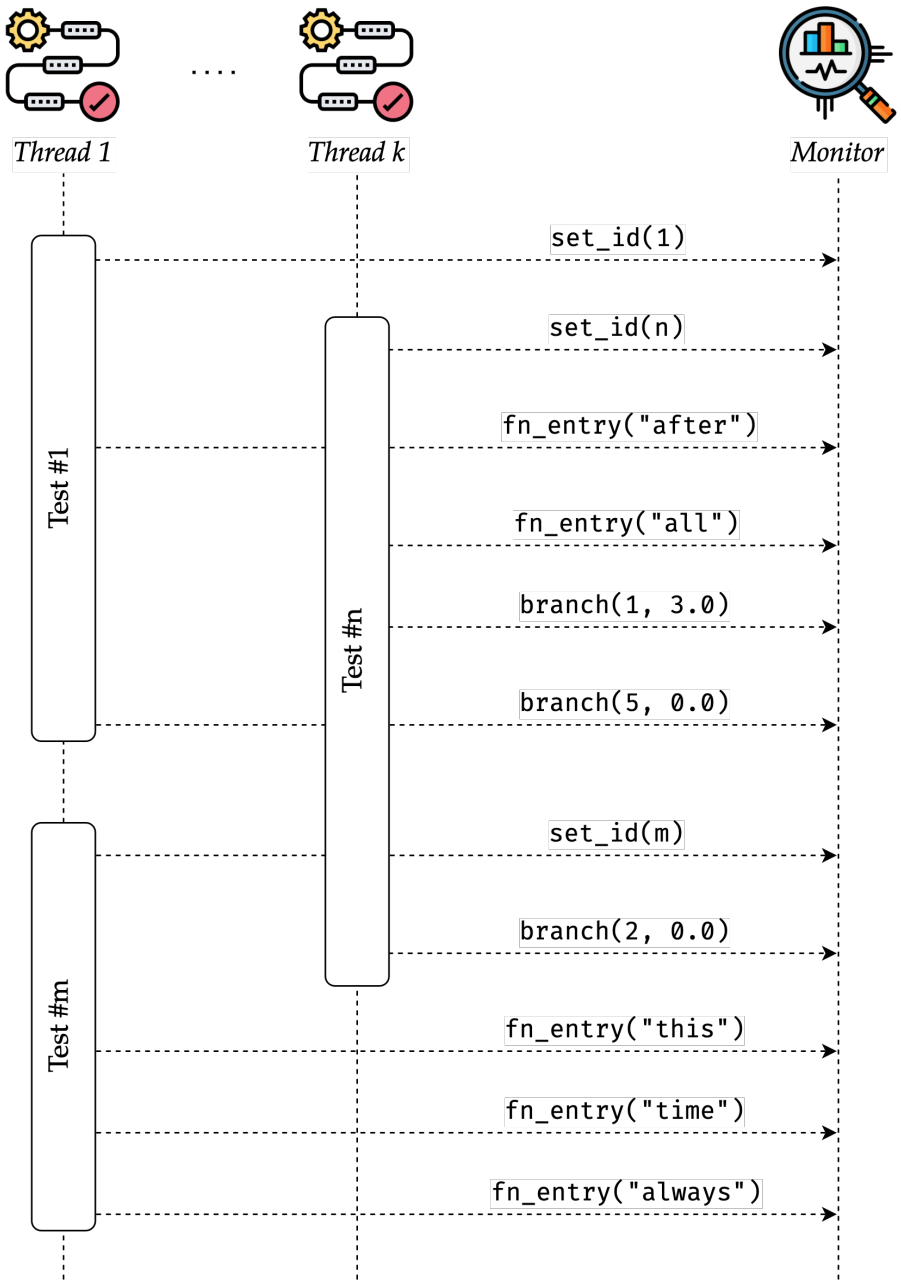
6.6. Test Execution

With Cargo, Rust provides a build system and a testing framework out-of-the-box. In Java, which is a managed language [47], generated tests can be executed directly using Reflection and bytecode instrumentation, and coverage information can be collected in the same runtime process. In Rust, we need to run the tests "traditionally", i.e., synthesize them into Rust source code, write into the approapriate source file of the SUT, compile, and execute. Due to incremental compilation, rustc generally only needs to recompile the changed test modules. Nevertheless, this introduces a non-negligible overhead that RustyUnit tries to minimize by compiling all tests in a population at once and concurrently executing them.

Cargo's test framework is called on the SUT to run the tests; it does so in parallel by default. To establish the mapping between each individual test and the corresponding execution traces, each test sets a `thread_local` ID at the beginning of its execution, which the instrumentation instructions exploit to report the correct "author" of coverage data.

As shown in Figure 6.5, an instrumented SUT reports important events during its execution to RustyUnit's monitor. Those are, for example, an entrance of a function or an execution of a decision branch. For the latter, the branch distance calculated on-the-fly is passed along with other data. RustyUnit's monitor is initialized before the test threads are started and can either write collected traces to files or to the Redis in-memory storage.

Figure 6.5. An example concurrent execution of multiple tests with a monitor



7

Evaluation

We provide an empirical study on our search-based tool RustyUnit for automatic test case generation. We start by our reserch questions in Section 7.1. Thereafter, we define the criteria for selecting case subjects for evaluation and present our case study in Section 7.2 and explain the environment setup we used in Section 7.3. In Section 7.4 we present the results and discuss the challenges of applying search-based testing to Rust in Section 7.5. Finally, we conclude this chapter with a discussion of the threats to validity in Section 7.6.

7.1. Research Questions

We now define questions that guide our evaluation. To determine how well our search-based approach performs with respect to achieved code coverage, a set of experiments shall be conducted on a set of open-source Rust programs. Since our implementation has still some limits regarding the language features of Rust, we first want to evaluate the performance of the algorithms on the *ToyCrate*, which is a small library written by ourselfs. *ToyCrate* only exploits Rust features that our analysis implementation supports but still has a complex structure, e.g., nested control flow. We provide the details of its implementation in Section 7.2.

The experiments should answer the following questions:

RQ1: How does the GA of RustyUnit compare to random search on supported language features?

RQ2: Do the comparison results generalize to real-world open-source crates?

RQ3: What are the characteristics of the generated tests?

7.2. Case Study Subjects

In this section, we shall describe the case study subjects we used for our experiments. To this end, we first give an overview of *ToyCrate*'s internals, and then we describe the collection of real-world open-source Rust crates that we selected for the evaluation.

ToyCrate

To conduct experiments whose results do not depend on the technical limitations of our approach, we created *ToyCrate*, an example crate which contains multiple structs. Its implementation shall mimic complex software with nested and dependent structure, while avoiding overly complex traits and generics. Listing 7.1 demonstrates one of *ToyCrate*'s structs and its implementation. It features nested control flow which poses a non-trivial search problem for random search as well as for our GAs.

Listing 7.1 One of the structs that *ToyCrate* provides

```
struct ParryHotter;

impl ParryHotter {
  fn alohomora(x: i32, y: i32, z: i32, b: i32) {
    if x == 7 {
      if y < 13 {
        // ...
      } else {
        // ...
      }
    } else if y == 10 {
      if y ⩾ 3000 {
        // ...
      } else {
        // ...
      }
    } else {
      // ...
    }
  }
}
```

Open-Source Crates

For the evaluation on real-world subjects, we randomly chose 6 open-source crates from the Cargo's crate registry. This results in a total of 2,965 functions and 12,842 MIR-level branches.

The libraries were chosen with respect to their testability. For experiments, it is necessary that the units are testable without complex interactions with external resources (e.g., databases, networks and filesystem) and are not multi-threaded. In fact, each experiment should be run in an independent way, and there might be issues if automatically generated test cases do not properly de-allocate resources. Notice that this is a problem common to practically all automated testing techniques in the literature. We also ignored crates that used native

Case Study	Version	LOC	Functions	Branches
ToyCrate	0.1	111	41	88
time	0.3.7	5123	1158	1147
gamie	0.7.0	328	116	594
lsd	0.21.0	3151	654	5609
humantime	2.1.0	414	87	437
quick-xml	0.23.0	3408	832	2025
tight	1.0.1	921	118	3030
$\sum$		12653	3006	12930

Table 7.1. Number of lines of code, functions, and MIR-level branches in the case study subjects

features such as foreign function interface. Table 7.1 summarizes the properties of these case study subjects in terms of the number of functions, branches, and lines of code. We calculated these metrics using Mozilla's *rust-code-analysis* tool [8]. By *functions* is meant the total number of associative functions, loose functions, and closures. *LOC* is the number of logical lines, i.e., statements, in the respective crate as specified by the *LLOC* metric from rust-code-analysis. The tool does not yet support the calculation of the number of branches for Rust, which is why we calculated this metric during the MIR analysis of the case study subjects. The *Branches* column shows the number of branching edges within the MIR of the respective crate.

7.3. Setup

As witnessed in Section 5.2, search algorithms are influenced by a great number of parameters. For many of these there are best practices. We chose many of them based on the experience of other search-based tools like EvoSuite [36] and Pynguin [62]. Altough longer test cases are better in general, we limited the length of test cases to $L = 100$ because we experienced this to be a suitable length at which the test case execution does not take too long, altough the initial test cases are generated with only 30 statements each. The population size for the GA was chosen to be 50. Additionally, we evaluted two versions of the GA of RustyUnit, seeded and non-seeded, also called vanilla, to analyze the impact of the seeding strategies that we described in Section 5.5.

Search algorithms are often compared in terms of the number of fitness evaluations or generated chromosome samples. In our evaluation, the algorithms that we evaluate have the same budget of successfully executed chromosomes, and the performance of the algorithms is compared using the code coverage of test cases that remain after the algorithms have used up the budget. This is due to the fact that some of the generated populations might contain uncompilable tests which

we cannot identify easily and therefore, need to throw away the whole population to recover.

As discussed by Ali et al. [1], a search algorithm should always be compared against at least random search in order to check that the algorithm is not simply successful because the search problem is easy. For evaluation, we implemented a random search algorithm in RustyUnit, which generates random test cases in a manner similar to the initial population in the genetic algorithm of RustyUnit, altough it does not apply optimizations like recombination and mutation. It also exploits an archive and stores test cases that execute previously uncovered branches in a SUT. The random search approach in our evaluation uses the same probability parameters as those set for RustyUnit. The intuition is that RustyUnit's GA will at least provide better test cases in terms of code coverage since the basis for generation is the same. The representations of our genetic algorithm and random search are constructed similarly (e.g., a chromosome is a single test in both cases); thus, such a comparison is a reasonable budget metric. This means that the search is performed until k generated test cases have been successfully executed as part of the fitness evaluations. In our experiments, we chose $k = 5000$. The stopping condition applies for both algorithms. For each case study subject and each algorithm, we ran the experiments 30 times with different seeds for the random number generator to minimize the influence of randomness.

For some crates, also those in our case study, the execution time of the generated tests may depend on the input data that the respective tests use to execute functions of the SUT; for instance, when a loop uses a function parameter as its upper bound. For randomly generated large numbers, the execution of such tests can therefore have a significant impact on the total time, especially since we generate thousands of tests. To prevent long-running tests, we set a hard timeout of 3 seconds for the execution of a single test case using the *timeout* attribute from the *ntest* [1] crate. If a test exceeds the timeout, it gets aborted and its coverage is recorded only until that point.

Rust does not incorporate the concept of exceptions; instead, the language employs the `Result` type for recoverable errors and crashes, or panics, on unrecoverable errors. `Result` is an enum and given an instance of it, we can check whether its wraps a true value or some error instance. The tests automatically generated by RustyUnit can fail for various reasons, e.g., because the arguments generated for a function call result in an integer overflow, a function calls `panic!()` explicitly under some condition, or a statement tries to unwrap an erroneous `Result`. Unfortunately, we cannot differentiate automatically what exactly the reason for a fail was, so failing tests are an obstacle. Since we explicitly try to cover all basic blocks, we want to have those test cases

[1] `https://web.archive.org/web/20220522213803/https://crates.io/crates/ntest`

in the final test suite and compute their coverage, too. Therefore, we exploit Rust's *should_panic* attribute for failing tests, which inverts the result of a test, i.e., a test passes if it panics, otherwise it fails.

7.4. Results

In the following we analyse the benchmark results in order to answer our research questions and test our hypothesis that the algorithms yield significantly different results. RustyUnit, as other search-based testing tools, relies on randomized algorithms, which are inherently affected by chance. Running a randomized algorithm twice will most likely produce different results. It is essential to use rigorous statistical methods to properly analyze the performance of randomized algorithms when we need to compare two or more of them. In this work, we follow the guidelines described by Arcuri and Briand [5]. Since comparisons with simpler alternatives (at a very minimum random search) is a necessity when one proposes a novel randomized algorithm or addresses a new software engineering problem [1], statistical testing should be part of the according work, which reports such empirical study. For each of the 6 crates, we ran RustyUnit against the random search to compare their achieved code coverage.

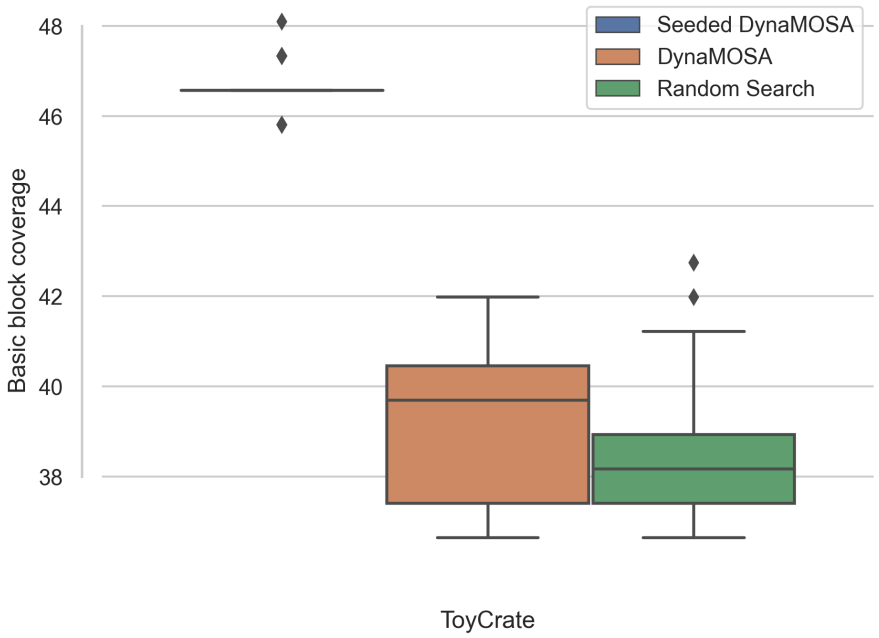
In particular, when basic block coverage values were compared, statistical difference has been measured with the Mann-Whitney U test. To quantify the improvement in a standadized way, we used the unparameterized Vargha-Delaney $\hat{A}_{12}$ effect size test. In our context, the $\hat{A}_{12}$ is an estimation of the probability that, if we run RustyUnit, we will obtain better coverage than running the random search strategy. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that, in *all* of the 30 runs of RustyUnit, we obtained coverage values higher than those obtained in *all* of the 30 runs of the random search strategy.

7.4.1. RQ1: Evaluation on ToyCrate

Figure 7.1 provides an overview of the coverage that the three algorithms achieved on the simple *ToyCrate* project. Obviously, seeded DynaMOSA outperforms the other algorithms and achieves an average coverage of 47%. The result is not astonishingly high but it's obvious that the algorithm successfully exploits the available domain knowledge to boost the search. Moreover, the test cases produced by seeded DynaMOSA not only achieve better coverage but also are shorter than those of other algorithms, i.e., they are 29.5 statements long on average.

Meanwhile, vanilla DynaMOSA did not exceel significantly and only achieved an average coverage value of 39%, which is similar to random search. However, the GA still generated shorter test cases than random search, with an average size of 32.8. Table 7.2 summarizes the the av-

Figure 7.1. Basic block coverage results for ToyCrate



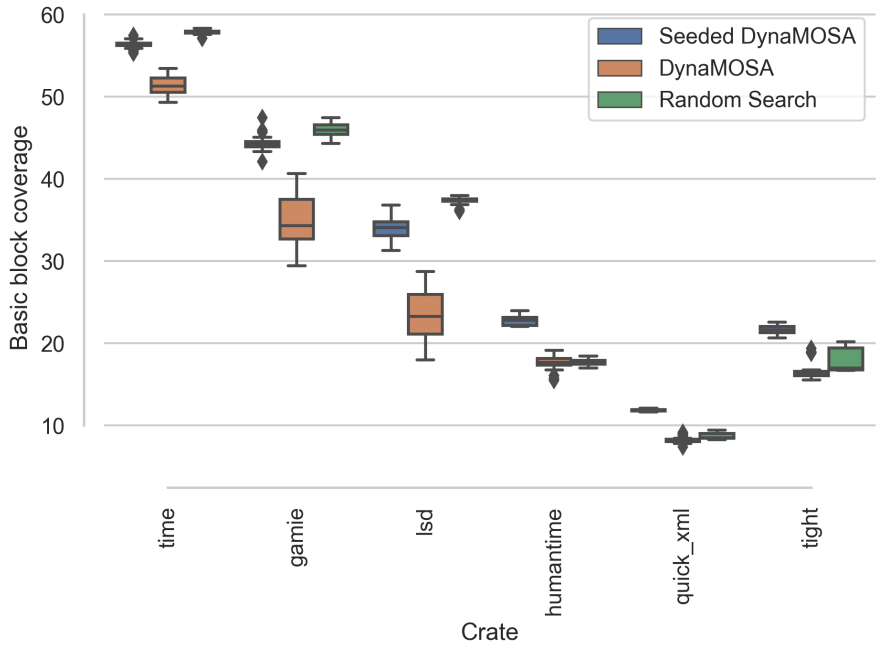
Metric	RS	DynaMOSA	$\hat{A}_{12}$	p-value
Coverage	0.38	0.47(0.39)	**1.0**(0.64)	**< 0.001**(0.05)
Length	36.9	29.5(32.8)	**0.95**(**0.76**)	**< 0.001**(**< 0.001**)

Table 7.2. The table reports the average basic block coverage and test case length obtained by random search and DynaMOSA with seeding and without seeding (in brackets) for *ToyCrate*. Effect sizes and p-values are in bold when the p-values are lower than 0.05

eraged results along with the effect sizes and p-values of the Mann-Whitney U test. We compare the GAs with random search. According to that, seeded DynaMOSA achieved better coverage results than random search in 100% of the cases and produced shorter tests in 95% of the cases, while its vanilla variant only yielded shorter test cases than random search in 76% of the cases. Its p-value for the code coverage indicates that the achieved coverage is not significantly different to that of random search, so we cannot reject the Ho hypothesis under these circumstances.

Summary (RQ1): Seeded DynaMOSA outperforms the other algorithms regarding both the achieved basic block coverage and the size of the tests generated. Vanilla DynaMOSA produces statistically similar results to random search, although it yields significantly shorter test cases.

Figure 7.2. Basic block coverage results per crate



The boxplot in Figure 7.2 compares the actual obtained basic block coverage values over 30 runs of RustyUnit's GAs and random search on the 6 open-source crates. In total, the coverage improvement of RustyUnit using the seeded DynaMOSA ranged up to 5% than that of random search. Meanwhile, vanilla DynaMOSA performed worse on every crate in the case study. Calculated on all the crates we evaluated, seeded DynaMOSA obtained an average coverage of 32%, whereas random search and vanilla DynaMOSA obtained 31% and 23%, respectively. The overall coverage results are not particularly high mostly due to traits that we cannot handle or types that are too advanced for our primitive analysis. For instance, we do not call functions that derived traits provide, e.g., `Serialize` and `Deserialize` by the *serde* crate, which is a wide-spread (de-)serialization library found in almost any mature crate, or *snafu*'s derived functionality, which provides domain-specific error types. Other than that, seeded DynaMOSA missed a few functions which random search did not miss; for example, the `new` functions of some structs, which, once called, get covered completely due to branch-less structure. Same limitations apply to the other crates.

It seems that at this point, a higher number of hit functions still plays a greater role than local search of uncovered decision branches. This is also indicated by the fact that the seeded GA performs much better than its vanilla variant. Our seeding strategy incorporates sampling all possible methods from the SUT one-by-one and usage of the SUT's

constant pool. However, the first three crates hardly contain any constants RustyUnit can practically leverage. For instance, *lsd* contains many string comparisons, as the function `from_arg_matches` suggests in Listing 7.2.

Listing 7.2 An example test that invokes `FooBar::foo`

```
fn from_arg_matches(matches: &ArgMatches)
    -> Option<Self> {
  if matches.is_present("tree") {
    Some(Self::Tree)
  } else if matches.is_present("long")
    || matches.is_present("oneline")
    || matches.is_present("inode")
    || matches.is_present("context")
    || matches!(
      matches.values_of("blocks"),
      Some(values) if values.len() > 1
    )
  {
    Some(Self::OneLine)
  } else {
    None
  }
}
```

In theory, our constant pool seeding should exceel at functions like that one and eventually cover all branches easily. However, in reality, `ArgMatches` is neither part of *lsd* itself, nor can RustyUnit generate its fields. It is rather a datatype from the *clap* crate, a command line argument parser, which *lsd* depends on.

The squeezed boxes in the Figure 7.2 clearly indicate that the two better performing algorithms have reached the technical limit of our implementation in quite every run for each crate. However, random search has a slight lead only for the first three crates, i.e., *time*, *gamie*, and *lsd*, while seeded DynaMOSA dominates the coverage for the remaining three ones, that is, *humantime*, *quick_xml*, and *tight*. And even vanilla DynaMOSA has at least caught up at this point. Table 7.3 provides a possible explanation for this observation. It summarizes the properties of the CDGs that we extracted from the case study subjects with respect to their number, the number of targets, and the average depth of the targets. *#Targets* refers to the overall number of basic blocks in the respective crate, while *Depth in CDGs* refers to the distance of a basic block to the respective root within its CDG. The average depth of a CDG target is 1.27. Keep in mind that we did not include *ToyCrate*'s data in the calculation to better illustrate a possible reason for those coverage values. An average depth value less than 1 indicates that there are functions that do not contain any blocks at all. As one

can see, the average depth values of *time* and *gamie* are nearly 1, which is a sign that most of their functions are shallow and hardly contain any nested logic, i.e., most basic blocks are control dependent on the root of the respective CDG. The phenomenon especially involves generated functions, e.g., derived trait implementations. In turn, a shallow function code structure means that it is sufficient to call it once to reach all basic blocks, which is why random search achieves decent results on the first three crates. That also means that RustyUnit cannot develop its full potential and employ guided search as it is mainly dependent on branch distance at most, if at all, which, as explained in Section 5.3, can only be 0 or 1 for non-numerical data. Thus, RustyUnit might degenerate to random search in such scenario. In fact, as we see, for the first three crates, it is even worse than that. Whereas random search continues to generate new test cases independently and hits more uncovered functions faster, the genetic algorithm tries to surgically improve test cases that often cannot be improved anyway.

However, things look differently for *humantime*, *quick_xml*, and *tight*. *humantime* and *quick_xml* feature a greater control dependence depth, which is where both genetic algorithms start to shine at, as they now can additionally leverage approach level for the fitness calculation. The average depth value of *quick_xml* is relatively low, though. At this point, however, our seeding strategy helps the GAs perform successfully. *quick_xml* is an XML parser library, whose API makes certain assumptions about the textual input it receives. For instance, consider the implementation block for the `new` function of the `BangType` enum in Listing 7.3, which, based on the input byte, decides which variant to return. RustyUnit was able to extract the constant byte values used in the function and inserted one of them in the test 4410.

Listing 7.3 Enum `BangType` is part of the *quick_xml* crate

```
impl BangType {
  fn new(byte: Option<u8>) -> Result<Self> {
    Ok(match byte {
      Some(b'[') => Self::CData,
      Some(b'-') => Self::Comment,
      Some(b'D') | Some(b'd') => Self::DocType,
      Some(b) => return Err(Error::UnexpectedBang(b)),
      None => return Err(
        Error::UnexpectedEof("Bang".to_string())
      ),
    })
  }
}

#[test]
#[timeout(3000)]
```

```
fn rusty_test_4410() {
  rusty_monitor::set_test_id(5410);
  let mut u8_0: u8 = 91u8;
  let mut option_2: Option<u8> = Option::Some(u8_0);
  let result_0: Result<BangType>
      = BangType::new(option_2);
}
```

The byte 91 is the ASCII value for '['. That is, RustyUnit successfully covered one of the branches. That one is quite simple, though. Listing 7.4 demonstrates another case of RustyUnit being able to extract and use a constant value successfully. The function `local_name` returns its name after stripping the namespace, which is separated with a double colon usually. `memchr` is a function that returns an `Option` with the index of the first occurence of a symbol in a string. If the `name` does not have a namespace, `local_name` just returns the `name` as is, otherwise it strips the namespace by returning a slice starting at `i + 1`.

Listing 7.4 Struct `BytesStart` is part of the *quick_xml* crate

```
impl<'a> BytesEnd<'a> {
  pub fn local_name(&self) -> &[u8] {
    let name = self.name();
    memchr::memchr(b':', name)
        .map_or(name, |i| &name[i + 1..])
  }
}

#[test]
#[timeout(3000)]
fn rusty_test_3247() {
    rusty_monitor::set_test_id(3247);
    let mut str_0: &str = "Attr::Empty";
    let mut str_0_ref_0: &str = &mut str_0;
    let mut bytescdata_1: BytesCData
        = BytesCData::from_str(str_0_ref_0);
    let mut cow_0: Cow<[u8]>
        = BytesCData::into_inner(bytescdata_1);
    let mut bytesend_0: BytesEnd
        = BytesEnd {name: cow_0};
    let mut bytesend_0_ref_0: &BytesEnd = &bytesend_0;
    let mut u8_slice_1: &[u8]
        = BytesEnd::local_name(bytesend_0_ref_0);
}
```

In test 3247, RustyUnit invokes the function `local_name` and builds a dependency sequence, which initially uses a constant string "Attr::Empty" in Line 13. RustyUnit again extracted the string value

Case Study	#CDGs	#Targets	∅ Target Depth in CDGs
ToyCrate	12	131	1.98
time	479	2923	1.0
gamie	262	1265	0.99
lsd	453	5539	1.27
humantime	70	848	1.48
quick-xml	305	2565	1.20
tight	99	2518	1.70
∑	1668	15658	∅ 1.27

Table 7.3. For each crate, the table reports the number of extracted CDGs, the sum of all targets, and the average depth of those targets within the respective CDG

out of the constant pool of the SUT. Since the string contains a colon, the execution hits the *true* case when invoking `local_name` in Line 23. In summary, despite the fact that the crate *quick_xml* does not feature particularly high control dependence degree, RustyUnit's seeded GA can still outperform random search using the seeding strategy.

To better understand whether RustyUnit's algorithm generally performs better, we provide statistical results in Table 7.4. The table presents the $\hat{A}_{12}$ values with respect to the basic block coverage we obtained for the crates in the case study. We report the statistics of both, seeded and vanilla version of the GA (in brackets) in comparison to random search. For instance, for *tight*, the $\hat{A}_{12}$ value of 0.96 means that the seeded DynaMOSA obtained a higher coverage in 96% of the cases. The table also provides results of the Mann-Whitney U test that we conducted per crate with the H0 hypothesis that each pair of the algorithms evaluated do not differ in terms of achieved coverage. We obtained p-values lower than the traditional $\alpha = 0.05$ for all crates, which means that the coverage differences over 30 executions are statistically significant. The seeded DynaMOSA could only achieve better effect size for 3 out of the 6 crates. However, Arcuri [5] suggests that the significance value α is much less of importance in the context of SBST; one shall state all the results regardless of their significance and let the reader decide for theirself to what extent a particular algorithm suits their needs. Statistical evaluation and practical, "felt" results can differ greatly for randomized algorithms. As for the vanilla GA, it cannot keep up with random search, which in general covers more basic blocks by running a new uncovered function than the GA can by covering few branches in the same time.

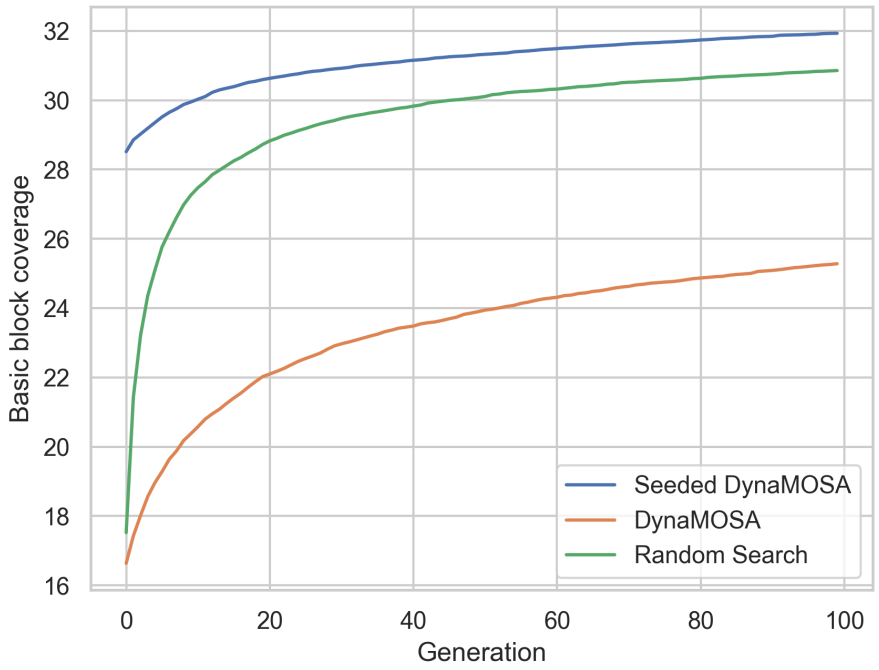
Regarding the statistical results, in theory, a t-test would be possible, too; however, it requires a normal distribution of the experiment results, which is not given for search-based test generation in general. Furthermore, we cannot determine the true mean and variance values since the search could run for an infinite period of time continuously generating better results, provided that the 100% coverage is not

Case Study	RS	DynaMOSA	$\hat{A}_{12}$	p-value
time	0.58	0.56(0.51)	0.11(0.0)	$<$ 0.001 ($<$ 0.001)
gamie	0.46	0.44(0.34)	0.10(0.0)	$<$ 0.001($<$ 0.001)
lsd	0.37	0.35(0.23)	0.22(0.0)	$<$ 0.001($<$ 0.001)
humantime	0.18	0.23(0.18)	1.0(0.52)	0.0($<$ 0.001)
quick-xml	0.09	0.12(0.08)	1.0(0.17)	0.0($<$ 0.001)
tight	0.18	0.22(0.17)	0.96(0.12)	0.0($<$ 0.001)
Average	0.31	0.32(0.25)	0.52(0.13)	

Table 7.4. For each crate, the table reports the average basic block coverage obtained by random search and DynaMOSA with seeding and without seeding (in brackets)

reached. Practically, the search is limited by the computational and memory capacity of the machine which the experiments run on, but it is only an artificial limit.

Figure 7.3. Average basic block coverage development over generations



All algorithms only achieve moderately high coverage up to 58%. The main reason is that our analysis of the possible function invocations and types is still very limited. We cannot execute functions of many generic traits since those either require some advanced type and constraint analysis, or parameters that we cannot instantiate. For instance, in case of `Debug`, which is one of the most often implemented traits, its only method `fmt` requires an argument of type `std::fmt::Formatter` that RustyUnit is not able to create using the

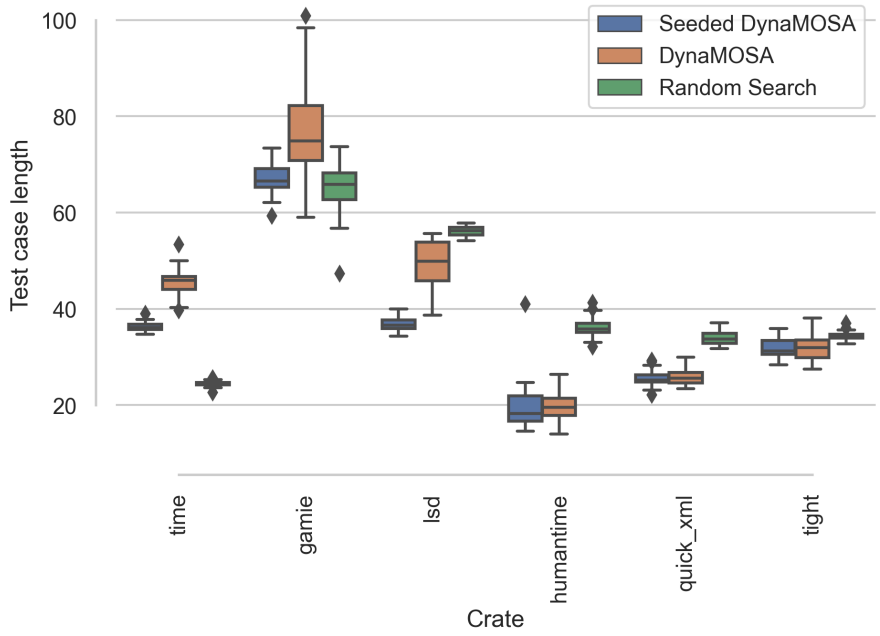
current implementation state. Usually, the parameter is provided automatically by the compiler during macro expansion when we print a certain type, e.g., with `print!()`. With these limitations given, currently, it is not possible to achieve 100% of code coverage for non-trivial data types.

Figure 7.3 illustrates the development of the average basic block coverage over all crates over the available 100 generations. For random search, we split the generated test cases into chunks to match the progress of the two other algorithms. The line plot clearly indicates that the seeding strategy chosen has great impact on the performance of the GA. It also outperforms random search on average. In summary, we can answer the second research question as follows:

Summary (RQ2): Despite the shallow code structure of the crates used, which prevent RustyUnit to unleash its full potential, it still achieves a comparable coverage to random test generation. For some crates, it achieves a significantly better coverage. Thus, we can state, that the advanced performance of the seeded DynaMOSA achieved on *ToyCrate* can be partially generalized to open-source crates. However, at the same time, vanilla DynaMOSA produced worse results compared to those of *ToyCrate*.

7.4.3. *RQ3: Characteristics of Generated Tests*

Figure 7.4. Test case length results per crate



In this section, we analyse the impact of the selected algorithm on the length of the generated test cases and try to answer RQ3, again using the per-crate Mann-Whitney U p-values and $\hat{A}_{12}$ effect sizes. Figure 7.4 illustrates the average test lengths of the tests produced by random search and the genetic algorithms of RustyUnit over the 30 runs. For the GAs, we observe a positive effect on crates which they also achieved high coverage for, i.e., *humantime*, *quick_xml*, and *tight*. Additionally, the genetic algorithms, especially the seeded variant, produce significantly shorter tests cases for *lsd*. Vanilla DynaMOSA yielded greatly scattered results for *gamie*, though, in contrast to the other algorithms. The reason for this behavior can be partly found in the structure of *gamie* itself. The crate provides implementations for board games, which it models using arrays, for instance:

```
pub struct Reversi {
  board: [[Option<Player>; 8]; 8],
  next: Player,
  status: GameState,
}
```

The board is an array of arrays of `Option` instances that wrap a `Player`. When some of the algorithms tries to instantiate the struct directly, it consequently needs to define all 64 `Option` instances, each as a distinct statement. This is what the other algorithms do, too. However, unlike seeded DynaMOSA, the vanilla variant seems to hit the struct instantiations more often due to a much smaller range of options given without seeding. It might happen once in a while, but one time is one too many, given that it generates 64 statements + 9 array declaration statements + the instance of the game itself.

Regarding *time*, both genetic algorithms seem to get stuck in generating functions that deal with enums sometimes. Many of the tests contain repetitive definitions and invocations, demonstrated in the following excerpt:

```
#[test]
#[timeout(3000)]
fn rusty_test_2501() {
  // ...

  let mut padding_8: Padding = Default::default();
  let mut padding_8_ref_0: &Padding = &mut padding_8;
  let mut padding_9: Padding = Padding::Optimize;
  let mut padding_9_ref_0: &Padding = &mut padding_9;
  let mut padding_10: Padding = Default::default();
  let mut padding_10_ref_0: &Padding = &mut padding_10;
  let mut padding_11: Padding = Padding::Optimize;
  let mut padding_11_ref_0: &Padding = &mut padding_11;
  let mut padding_12: duration::Padding
```

Case Study	RS	DynaMOSA	$\hat{A}_{12}$	p-value
time	24.4	36.2(45.6)	**0.0**(**0.0**)	**< 0.001**(**< 0.001**)
gamie	65.0	67.0(77.3)	0.38(**0.16**)	0.11(**< 0.001**)
lsd	56.1	36.8(49.3)	**1.0**(**0.96**)	**< 0.001**(**< 0.001**)
humantime	36.1	19.8(20.0)	**0.97**(**1.0**)	**< 0.001**(**< 0.001**)
quick-xml	33.9	25.6(25.9)	**1.0**(**1.0**)	**< 0.001**(**< 0.001**)
tight	34.4	31.8(31.8)	**0.87**(**0.84**)	**< 0.001**(**< 0.001**)
Average	41.7	36.4(41.5)	0.42(0.34)	

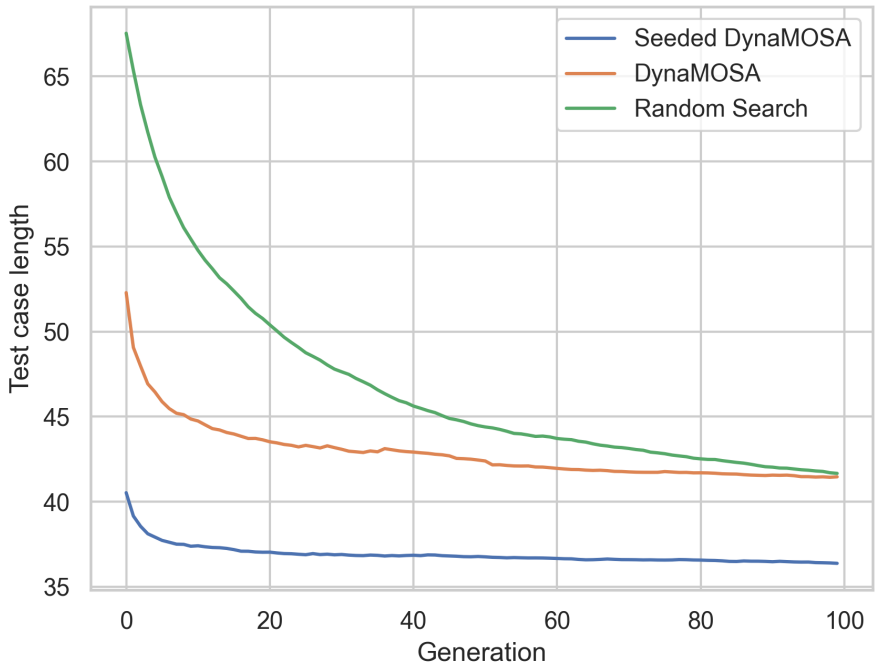
Table 7.5. For each crate, the table reports the average test case length obtained by random search and DynaMOSA with seeding and without seeding (in brackets). Effect sizes and p-values are in bold when the p-values are lower than 0.05

```
15          = Default::default();
16      let mut padding_12_ref_0: &duration::Padding
17          = &mut padding_12;
18      let mut padding_13: Padding = Padding::Optimize;
19      let mut padding_13_ref_0: &duration::Padding
20          = &mut padding_13;
21      let mut option_0: Option<Ordering>
22          = PartialOrd::partial_cmp(
23              padding_13_ref_0, padding_12_ref_0
24            );
25      let mut option_1: Option<Ordering>
26          = PartialOrd::partial_cmp(
27              padding_11_ref_0, padding_10_ref_0
28            );
29      let mut option_2: Option<Ordering>
30          = PartialOrd::partial_cmp(
31              padding_9_ref_0, padding_8_ref_0
32            );
33      let mut option_3: Option<Ordering>
34          = PartialOrd::partial_cmp(
35              padding_7_ref_0, padding_6_ref_0
36            );
37  }
```

Table 7.5 demonstrates the average length value for all algorithms, as well as the effect sizes and p-values. The difference in length is statistically significant for all crates except for *gamie*, which the seeded DynaMOSA produced similar test case length for, as random search did. In six out of those, RustyUnit was able to generate shorter tests than random search. Random search could only yield shorter test cases for *time* and *gamie* and outperformed both GAs.

Summary (RQ3): The GAs of RustyUnit generate shorter test cases for most of the case study subjects.

Figure 7.5. Average test case length development over generations



7.5. Discussion

In this section, we shall discuss and summarize the challenges of applying search-based test generation to Rust as a programming language.

Traits

One of the most important aspects of Rust's type system and, at the same time, the biggest technical limitation of RustyUnit are traits, which can become very complex and require advanced analysis. Similar to interfaces in other programming languages, traits provide the ability of having multiple implementations of some protocol to interact with an object. However, compared to Java's interfaces, for instance, Rust's traits are more powerful. Java's interfaces can be generic, too, but a class can implement it only once for some generic or concrete type. In contrast, a data type in Rust can provide a range of different implementations for the same trait.

In Listing 7.5, struct `Int` with a generic type parameter implements the trait `Value`, which also features a generic type parameter `T` and declares a function `value(&self)`.

Listing 7.5 A struct providing multiple implementations for the same struct

```
1 struct Int<Bit> {
```

```
  value: Bit
}

trait Value<T> {
  fn value(&self) -> T;
}

impl Value<i32> for Int<i64> {
  fn value(&self) -> i32 {
    self.value as i32
  }
}

impl Value<i32> for Int<i32> {
  fn value(&self) -> i32 {
    self.value * 2
  }
}
```

This has an important implication for the test generation problem. While we can assume to be able to invoke an interface's method on an object of type implementing the interface, in Rust, generally, we cannot. In the example, we defined the function `value` only for types `Int<i32>` and `Int<i64>`. Calling the trait function on versions of `Int` would inevitably lead to compile errors.

Moreover, we can also vary the generic type of the trait itself, which would change the return type of the function:

```
// ... other implementations

impl Value<i16> for Int<i32> {
  fn value(&self) -> i16 {
    self.value as i16
  }
}

fn main() {
  let i = Int{value: 32i32};
  // Compile error: Which implementation is meant?
  let value = i.value();
}
```

Now, there is some ambiguity in case we call the function of the trait in Line 12 which we shall resolve either by declaring the variable's type explicitly or naming the implementation of the trait as in Listing 7.6. Thus, during the test generation, one not only needs to parse trait implementations but also detect which type variants the traits belong to. However, this task can get even more complicated. Assume that we

do not want to implement the trait, e.g., `Value<i64>`, for each possible primitive type.

Listing 7.6 Trait implementation for generic `Int`

```
1  // ... other implementations
2
3  impl<T: Into<i64> + Copy> Value<i64> for Int<T> {
4    fn value(&self) -> i64 {
5      self.value.into()
6    }
7  }
8
9  fn main() {
10   let i = Int{value: 32i32};
11   let i64_value = <Int<i32> as Value<i64>>::value(&i);
12
13 }
```

Instead, in Listing 7.6 we can maintain a single implementation for all `Int<T>` which provides the function to all `Int` instances whose generic type implements `Into<i64>`, i.e., which can be casted to `i64` and is copyable. It has to be copyable since the function moves `value` out of the parent type. In Line 11, we specify which implementation of the trait shall be used to invoke the function `value`. At this point, to generate a test that covers the implementation in Line 5, a search algorithm needs to find a substitution for the generic type parameter that complies to the specified trait bounds.

A more general form of trait implementations are so-called *blanket trait implementations*. Those are implementations of a trait for all types or types that match some condition. In Listing 7.7, we implement `Value` for all types that implement the `Display` trait from the standard library and have a compile-time known size.

Listing 7.7 Trait implementation for generic `Int`

```
1  // ... other implementations
2
3  impl<T: std::fmt::Display + Sized> Value<String> for T {
4    fn value(&self) -> String {
5      format!("Value: {}", self)
6    }
7  }
8
9  fn main() {
10   let a = 10i32;
11   let str_value = a.value();
12   println!("{}", str_value);
13 }
```

With the blanket implementation in place, we not only can invoke the trait function on custom datatypes but also on types defined outside of our crate, e.g., standard library, as demonstrated in Lines 10-11 on a 32-bit integer. Blanket implementations are an important feature for high code coverage of generated tests since many APIs are designed to accept data types that the authors provide blanket implementations for.

In summary, to produce compilable test cases that at least can invoke all possible functions in a SUT and, moreover, achieve acceptable coverage results, one needs a sophisticated approach to model the underlying type system, analyze constraints, and map implementations to all appropriate datatypes.

Test Execution

Due to the static nature of Rust, we cannot "just run" the generated test cases at hand. With RustyUnit, we implemented the most trivial and straightforward possibility to execute generated tests: we convert them into string representation, sort the tests by appropriate Rust modules and append the stringified versions at the end of the source files. Finally, we compile the SUT with Cargo as one would do manually. Obviously, this is not the most efficient solution. Executing the first population takes much longer than the rest because the whole SUT including its dependencies needs to be compiled. Afterward, Rust's incremental compilation comes into play; thus, execution of subsequent populations of tests takes significantly less time since only the crate under test changes and needs recompilation. However, to achieve optimal results, we need thousands of tests. Moreover, since the algorithm is a probabilistic one, we need to repeat experiments many times to get more or less stable results usable for statistical evaluation. At this point, the execution duration issue becomes apparent and is the reason why we have not conducted more repetitions of our experiments. This aspect definitely needs to be considered when planning large experiments.

MIR

Another important point for a search-based testing is how Rust handles compound boolean expressions and short-circuit evaluation. Assume the function `foo` which returns some integer based on its inputs, as shown in Listing 7.8. The if condition is a compound boolean expression; thus, technically, the function features four branches, two for each sub-expression.

Listing 7.8 A function with a compound conditional expression

```
1 fn foo(x: i32, y: i32) -> i32 {
2   if x < 3 || y > 2 {
```

```
3       10
4     } else {
5       20
6     }
7   }
```

Listing 7.9 demonstrates how the compiler expands the branches. As in the original function, the compiler starts by examining the outcome of the first comparison in Lines 6 and 7. If the result of the evaluation is true, then the second sub-expression no longer plays a role for the control flow because of the logical *or* operator. Thus, the execution jumps to block `bb1`, otherwise to block `bb2`. Remarkable, in both cases the compiler defines the same variable in Lines 11 and 18, which `bb3` uses to determine the branch to take.

Listing 7.9 Expansion of the `foo`'s conditional expression at the MIR level

```
1  fn foo(_1: i32, _2: i32) -> i32 {
2    // Locals definition
3
4    bb0: {
5      _5 = _1;
6      _4 = Lt(move _5, const 3_i32);
7      switchInt(move _4) -> [false: bb2, otherwise: bb1];
8    }
9
10   bb1: {
11     _3 = const true;
12     goto -> bb3;
13   }
14
15   bb2: {
16     _7 = _2;
17     _6 = Gt(move _7, const 2_i32);
18     _3 = move _6;
19     goto -> bb3;
20   }
21
22   bb3: {
23     switchInt(move _3) -> [false: bb5, otherwise: bb4];
24   }
25
26   bb4: {
27     _0 = const 10_i32;
28     goto -> bb6;
29   }
30
31   bb5: {
```

```
32      _0 = const 20_i32;
33      goto -> bb6;
34    }
35
36    bb6: {
37      return;
38    }
39  }
```

Thus, the second comparison is performed at the MIR level in any case, but without causing possible side-effects. What's also important is that the value used in the second `SwitchInt` has multiple definitions in parallel blocks, which prevents us from tracing a meaningful distance value for the second `SwitchInt`. We cannot statically determine how `_3` came into play using RustyUnit's basic analysis. RustyUnit assumes that all variables have been defined sequentially and as consequence, cannot determine whether to trace `const true` or `Lt(_5, 3_i32)`. To handle such cases, we need to save the state of the execution in some way. One possibility is to trace all atomic statements and thus, all actual local values, to the monitor during execution. Hence, in case of a `SwitchInt`, one would only need to report the corresponding variable and let the monitor dynamically calculate the distance based on the recorded state. However, one has to consider that each call invocation is a terminator and thus a separate basic block; a great number of additional tracing blocks may lead to high execution overhead.

As an alternative, one might calculate local branch distances in-place as additional statements after the definition of relevant variables. For instance, we would insert the following statements in `bb1` and `bb2`:

Listing 7.10 Computing branch distances in-place within the MIR

```
fn foo(_1: i32, _2: i32) -> i32 {
  // ...

  bb1: {
    _3 = const true;
    // Distance to true branch:
    _10 = const 0.0_f64;
    // Distance to false branch:
    _11 = move _5 as f64;
    _12 = move const 3i32 as f64;
    _10 = Sub(move _12, move _11);
    goto -> bb3;
  }

  bb2: {
    _7 = _2;
    // x > y
```

```
    _6 = Gt(move _7, const 2_i32);
    // Distance to true branch: y - x + 1.0
    _8 = move _2 as f64;
    _9 = move const 2_i32 as f64;
    _10 = Sub(move _9, move _8);
    _10 = Add(_10, const 1.0_f64);
    // Distance to false branch: x - y
    _11 = Sub(move _8, move _9);
    _3 = move _6;
    goto -> bb3;
  }

  bb3: {
    switchInt(move _3)
      -> [false: bb10, otherwise: bb12];
  }

  // Trace a false branch hit
  bb10: {
    _19 = monitor::trace_branch_hit(const 2_u64,
                        const 3_u64,
                        const 5_u64) -> bb11;
  }

  // Trace distance to the true branch
  bb11: {
    _20 = monitor::trace_branch_distance(const 2_u64,
                        const 3_u64,
                        const 5_u64,
                        move _10,
                        const true) -> bb5;
  }

  // Trace a true branch hit
  bb12: {
    _21 = monitor::trace_branch_hit(const 2_u64,
                        const 3_u64,
                        const 4_u64) -> bb13;
  }

  // Trace distance to the false branch
  bb13: {
    _20 = monitor::trace_branch_distance(const 2_u64,
                        const 3_u64,
                        const 4_u64,
                        move _11,
```

```
                                const false) -> bb4;
    }

    // ...
}
```

In Listing 7.10, we identify that a `SwitchInt` depends on a value defined in two parallel blocks, so we compute the branch distance inplace right after the defintion of relevant locals. With the computation set up, we can inject tracing chains (`bb10`, `bb11`) and (`bb12`, `bb13`). The tracing chains now employ the statically known locals `_10` and `_11`, whose actual value is determined by the path executed.

Enums

Enumerations are first-class citizens in Rust's world. For instance, the two most prominent enums, `Option` and `Result` can be found just in any crate that uses standard library. In contrast to other languages, enums are more powerful. Their variants can wrap other datatypes or be structs themselves. Thus, a test generation approach must definitely take them into account to achieve optimal code coverage. Datatype wrapping is especially a challenge when looking for suitable generators for a datatype, as *GenObject* does in Algorithm 7, if a generator returns a datatype wrapped (possibly multiple times) into an enum. Simply checking equality of types will fail in those cases. In general, one can always create an enum instance "by hand" with random values; however, a generator possibly sets up a valid and correct state when returning an instance. Consider the implementation of `Minesweeper` in Listing 7.11. It is part of the *gamie* library, which provides implementations for basic games.

Listing 7.11 A code excerpt from *gamie*

```
struct Minesweeper {
  board: Vec<Cell>,
  height: usize,
  width: usize,
  status: GameState,
}

impl Minesweeper {
  pub fn new<R: Rng + ?Sized>(
    height: usize,
    width: usize,
    mines: usize,
    rng: &mut R,
  ) -> Result<Self, MinesweeperError> {
    if height * width < mines {
```

```
16          return Err(MinesweeperError::TooManyMines);
17      }
18
19      let board = iter::repeat(Cell::new(true))
20          .take(mines)
21          .chain(
22              iter::repeat(Cell::new(false))
23                  .take(height * width - mines)
24          )
25          .collect();
26
27      let mut minesweeper = Self {
28          board,
29          height,
30          width,
31          status: GameState::InProgress,
32      };
33      minesweeper.randomize(rng).unwrap();
34
35      Ok(minesweeper)
36  }
37 }
```

Given that the input values are appropriate, the `new` function creates a well-defined state of the board and returns the instance wrapped in an `Option`. Invoking functions on that instance has a higher chance of covering regular execution paths than on an instance that one created by setting its fields directly with random values, as data fields are often not independent and feature connected semantics. It is an open question how to access the wrapped instance in a generated test, though. `Option` and `Result` provide the `unwrap()` function, which returns the wrapped value. However, most enums do not. Usually, one accesses internals of an enum through pattern matching in `if` and `match` statements, which would make generated tests more complicated.

Tooling

Rust is a very young programming language, which implies that the variety of tools tailored for very specific tasks is not yet as rich as it is for other languages. For instance, there are no tools for program instrumentation such as Javassist [23] or ASM [16] for Java. Therefore, we needed to write our own compiler wrapper that parses crates' internals and injects atomic instructions for tracing. Given the size and complexity of the language, we had to limit it to a reasonable minimum. Thus, the prototype of RustyUnit does not support many languge features of Rust, like slices or lamdas.

Additionally, the situation regarding mocking frameworks for Rust is somewhat difficult. Mocking is widely used technique to eliminate dependencies and mimic their behavior in tests. At the time of writing, multiple frameworks already exist for Rust. They generate mocks at compile time for elements that one annotates with a special macro attribute, similarly to the `#[derive( ... )]` macro. However, they come with different limitations and can mostly only mock traits but not structs or enums. Nevertheless, trait mocks can still be useful, e.g., to generate instances of datatypes that we are not able to instantiate in the generated tests. It is an open question, how to incorporate mocks automatically without annotating the SUT manually.

7.6. Threats to Validity

In this section, we provide an overview of the threats to validity of our evaluation.

Internal Validity

In our experiments, we compare algorithms in terms of the achieved basic block coverage since the standard Rust coverage measuring tool *instrument-coverage* only supports statement and region coverage but not branch coverage. An alternative, *gcov*, instruments the LLVM IR in contrast to MIR instrumentation done by *instrument-coverage* and is imprecise in what it can measure. Therefore, we implemented our own coverage measurements by instrumenting the MIR and logging visited basic blocks. We carefully investigated the implementation to prevent possible errors.

External Validity

For our study, we used only 6 open-source crates for our evaluation. They are compact and, above all, have as few dependencies as possible, i.e., they stay self-contained. The crates do not interact with the environment, e.g., by means of I/O, nor do they use some sort of parallelism. Furthermore, the number of subjects and their intention is not diverse enough, so the results may not generalize to all Rust programs.

Construct Validity

Strictly speaking, some of the generated tests may not fall into the category of unit tests, as our approach cannot mock dependencies and instead generates real instances regardless of their origin and complexity. The testing frameworks for Rust are still very limited in their flexibility because mocks, like so many other things in Rust, are created using macros at compile time. Also, they can mostly only mock traits and no structs or enums. Furthermore, with our approach, we

cannot measure fault finding capability of the generated tests as our tool does not employ assertions, which is explicitly out of scope of this work. Finally, our approach exploits instrumentation and analysis of MIR, which is unstable and is subject to silent and breaking changes. That is, applying RustyUnit on Rust programs with future version of the Rust compiler might require some changes of the instrumentation technique used.

Conclusion

We conclude the work with a summary and outlook on possibilities for future work.

8.1. Summary

In this work, we presented RustyUnit, the first automated search-based test case generation framework for Rust that is available as an open-source tool, and showed that RustyUnit is able to emit test cases for Rust that cover non-neglectable parts of existing code bases. RustyUnit provides a GA-based and a random test generation approach, which we empirically evaluated on 6 open-source Rust projects. Our results confirm previous findings from the related work that a GA-based approach can outperform a random approach in terms of achieved coverage. We further showed that the availability of a seeding strategy greatly improves the effectivity and performance of our approach. However, in our experiments with the Rust programs, we learned that the strong type system poses many challenges to static analysis and wildly mutating tests produced bya GA. Also, we showed that the internal structure of Rust programs is somehow different to those of other languages in the sense that Rust functions do not feature greatly control-dependent blocks of code, which a GA usually exceels at. Finally, our investigations revealed a range of technical challenges for automated test generation for Rust, which provide great opportunities for further research, for instance, the generation of assertions, mocks, or the integration of type inference approaches.

8.2. Future Work

Solving a research problem almost always leads to new problems and further open questions. In the final section of this thesis, we want to discuss a few questions that arose during the work on the thesis. We leave them as future work because they are out of this work's scope.

We already stated in the evaluation that the whole process of generating tests for Rust with RustyUnit is relatively slow and cannot be applied easily to all crates of any size out in the wild. The execution time is tied to the execution of the Rust compiler and I/O operations,

which means that the cost of the runtime of the generated tests themselves is not the bottleneck in general. At this point, cache optimizations could be made or the way SUT is instrumented and tests are executed could be adjusted. Furthermore, the generated tests contain a lot of unnecessary statements even though RustyUnit eliminates some of them during search, which lead to the running, but more importantly comprehension overhead of a humble developer who would have to deal with them since the tests do not automatically employ assertions. This leads us to the other aspect of test generation: RustyUnit does not generate real test cases, but only program executions. A test oracle, which determines whether the result of an execution is correct and meets the expectations, is missing. Some test generation tools, such as EvoSuite, can automatically incorporate assertions, for example by using the recorded behavior of SUT for regression testing. In the future, a mechanism for automatically generated oracles could also be provided in RustyUnit.

Another aspect of test generation is the scope of the test cases. Unit testing, as the name suggests, refers to individual units. Traditionally, complex dependencies are replaced by mocks in unit tests, e.g., a database connection. As far as possible, RustyUnit exploits real data types, even if they originate in other units, which makes some test more integration than unit tests. Rust's ecosystem already offers many mock libraries, but unlike Java, they do not rely on runtime reflection, but on Rust's strong macro system and generate mocks at compile time. It would be interesting to extend RustyUnit's search-based algorithm with functional mocks to achieve potentially better coverage, such as Arcuri et al. [3] do for EvoSuite.

One further significant facet to consider is the execution environment. Real and complex software often interacts with its environment, for example by making I/O calls to the file system or establishing Transmission Control Protocol (TCP) connections. Lack of handling of the execution environment the SUT lives within is still one of the open problems in SBST [65] and poses many problems since running a program with automatically and randomly generated input data can have unpredictable effects, such as, for example, hard disk erasure. To avoid unwanted side effects, EvoSuite [40] exploits the Java Security Manager to prohibit most possible interactions of a SUT with its environment. KLEE [17] takes a different approach and redirects concrete calls, such as file system's `open()` and `read()`, to *models* that understand the semantics of the desired action well enough to generate the required constraints. A similar approach could be applied to Rust programs, i.e., transform the MIR of a SUT on-the-fly during the compilation in such way that environment interaction become harmless or even mock them.

As for seeding the initial population of test cases, there exist futher techniques that could be applied and evaluated for Rust, such as incor-

porating manually written unit tests. Often programs already contain test cases that do not cover the entire code but only the regular execution flow. An approach is to use those tests as part of the initial population to cover edge cases, for instance. However, the existing tests must first be converted into the representation that a GA uses. This means that they have to be parsed in the case of RustyUnit. Luckily, Rust already provides mature libraries for this purpose, e.g., *syn* [1].

Finally, it would be useful to use Rust's compiler analysis for more meaningful tests. RustyUnit is a compiler wrapper and can thus access all compiler internals during the analysis of the SUT. Advanced static analysis of the SUT, such as interprocedural search of the initialization of operands used in conditions, could help generate better tests with higher coverage.

[1] `https://web.archive.org/web/20220520083112/https://crates.io/crates/syn`

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, May 31, 2022 _____

Vsevolod Tymofyeyev

Acronyms

SUT System Under Test

GA Genetic Algorithm

EA Evolutionary Algorithm

MOSA Many-Objective Sorting Algorithm

DynaMOSA Many-Objective Sorting Algorithm with Dynamic target selection

WS Whole Suite

WSA Whole Suite with Archive

SBST Search-based Software Testing

SBSE Search-based Software Engineering

ATP Automated Theorem Prover

DSE Dynamic Symbolic Execution

IR Intermediate Representation

MOA Multi- and Many-Objective Algorithm

HIR High-level Intermediate Representation

THIR Typed HIR

MIR Mid-level Intermediate Representation

AST Abstract Syntax Tree

CFG Control Flow Graph

CDG Control Dependence Graph

API Application Programming Interface

NSGA-II Non-dominated Sorting Genetic Algorithm II

TCP Transmission Control Protocol

SSA Static Single Assignment

B

Bibliography

[1] S. Ali et al. "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation". In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 742–762.

[2] B. Anderson et al. "Engineering the Servo Web Browser Engine Using Rust". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 81–89.

[3] A. Arcuri, G. Fraser, and R. Just. "Private API Access and Functional Mocking in Automated Unit Test Generation". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 2017.

[4] A. Arcuri. "It really does matter how you normalize the branch distance in search-based software testing". In: *Software Testing, Verification and Reliability* 23.2 (2013), pp. 119–147.

[5] A. Arcuri and L. Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering". In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2011, pp. 1–10.

[6] A. Arcuri and G. Fraser. "On the Effectiveness of Whole Test Suite Generation". In: (2014), pp. 1–15.

[7] A. Arcuri and X. Yao. "Search based software testing of object-oriented containers". In: *Information Sciences* 178.15 (2008), pp. 3075–3095.

[8] L. Ardito et al. "rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes". In: *SoftwareX* 12 (2020), p. 100635.

[9] M. Asay. *Why AWS loves Rust, and how we'd like to help*. URL: `https://aws.amazon.com/de/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/` (visited on 06/28/2021).

[10] T. Ball and D. Jakub. "Deconstructing Dynamic Symbolic Execution". In: *NATO Science for Peace and Security Series, D: Information and Communication Security* 40 (2015), pp. 26–41.

[11] E. T. Barr et al. “Automatic Detection of Floating-Point Exceptions”. In: *SIGPLAN Not.* 48.1 (2013), pp. 549–560.

[12] B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[13] L. Bergstrom. *Google joins the Rust Foundation*. URL: `https://opensource.googleblog.com/2021/02/google-joins-rust-foundation.html` (visited on 11/08/2021).

[14] N. Bhattacharya et al. “Divide-by-Zero Exception Raising via Branch Coverage”. In: *Search Based Software Engineering*. Ed. by M. B. Cohen and M. Ó Cinnéide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 204–218. ISBN: 978-3-642-23716-4.

[15] P. Braione et al. “SUSHI: A Test Generator for Programs with Complex Structured Inputs”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE Computer Society, 2018, pp. 21–24.

[16] E. Bruneton, R. Lenglet, and T. Coupaye. “ASM: a code manipulation tool to implement adaptable systems”. In: *Adaptable and extensible component systems* 30.19 (2002).

[17] C. Cadar, D. Dunbar, and D. Engler. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs.” In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[18] C. Cadar and D. Engler. “Execution Generated Test Cases: How to Make Systems Code Crash Itself”. In: *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 2–23.

[19] C. Cadar et al. “EXE: Automatically Generating Inputs of Death”. In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008).

[20] J. Campos, A. Panichella, and G. Fraser. “EvoSuiTE at the SBST 2019 tool competition”. In: *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE Computer Society, 2019, pp. 29–32.

[21] J. Campos et al. “An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation”. In: *Search Based Software Engineering*. Cham: Springer International Publishing, 2017, pp. 33–48.

[22] J. Campos et al. “An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation”. In: *Search Based Software Engineering*. Cham: Springer International Publishing, 2017, pp. 33–48.

[23] S. Chiba. “Load-Time Structural Reflection in Java”. In: *ECOOP 2000 — Object-Oriented Programming*. Ed. by E. Bertino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 313–336.

[24] L. Clarke. “A System to Generate Test Data and Symbolically Execute Programs”. In: *IEEE Transactions on Software Engineering* SE-2.3 (1976), pp. 215–222.

[25] K. D. Cooper, T. J. Harvey, and K. Kennedy. "A simple, fast dominance algorithm". In: *Software Practice & Experience* 4.1-10 (2001), pp. 1–8.

[26] L. Davis. *Handbook of genetic algorithms*. Van Nostrand Reinhold, New York, 1991.

[27] M. D. Davis and E. J. Weyuker. "Pseudo-Oracles for Non-Testable Programs". In: *Proceedings of the ACM '81 Conference*. New York, NY, USA: Association for Computing Machinery, 1981, pp. 254–257.

[28] K. Deb and K. Deb. "Multi-objective Optimization". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Boston, MA: Springer US, 2014, pp. 403–449.

[29] K. Deb et al. "A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II". In: (2000), pp. 849–858.

[30] B. Doerr, C. Doerr, and F. Ebel. "From black-box complexity to designing new genetic algorithms". In: *Theoretical Computer Science* 567 (2015), pp. 87–104.

[31] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[32] A. Fioraldi et al. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 2020.

[33] G. Fraser and A. Arcuri. "1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite". In: *Empirical Software Engineering* 20.3 (2013), pp. 611–639.

[34] G. Fraser and A. Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (2014).

[35] G. Fraser and A. Arcuri. "Automated Test Generation for Java Generics". In: *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*. Cham: Springer International Publishing, 2014, pp. 185–198.

[36] G. Fraser and A. Arcuri. "Evolutionary Generation of Whole Test Suites". In: *2011 11th International Conference on Quality Software*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40.

[37] G. Fraser and A. Arcuri. "EvoSuite". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 416–419.

[38] G. Fraser and A. Arcuri. "EvoSuite at the SBST 2016 Tool Competition". In: *Proceedings of the 9th International Workshop on Search-Based Software Testing*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 33–36.

[39] G. Fraser and A. Arcuri. "EvoSuite at the SBST 2017 Tool Competition". In: *10th International Workshop on Search-Based Software Testing (SBST'17) at ICSE'17*. 2017, pp. 39–42.

[40] G. Fraser and A. Arcuri. "EvoSuite: On the Challenges of Test Case Generation in the Real World". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2013, pp. 362–369.

[41] G. Fraser and A. Arcuri. "The Seed is Strong: Seeding Strategies in Search-Based Software Testing". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 121–130.

[42] G. Fraser and A. Arcuri. "Whole Test Suite Generation". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291.

[43] G. Fraser, J. M. Rojas, and A. Arcuri. "EvoSuite at the SBST 2018 Tool Competition". In: *Proceedings of the 11th International Workshop on Search-Based Software Testing*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 34–37.

[44] G. Fraser and A. Zeller. "Mutation-Driven Generation of Unit Tests and Oracles". In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 278–292.

[45] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 213–223.

[46] D. Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM Comput. Surv.* 23.1 (1991), pp. 5–48.

[47] J. Gough. "Virtual machines, managed code and component technology". In: *2005 Australian Software Engineering Conference*. IEEE Computer Society, 2005, pp. 5–12.

[48] N. Gupta, A. Mathur, and M. Soffa. "Generating test data for branch coverage". In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2000, pp. 219–227.

[49] M. Harman et al. "Testability transformation". In: *IEEE Transactions on Software Engineering* 30.1 (2004), pp. 3–16.

[50] M. Harman and P. McMinn. "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search". In: *IEEE Transactions on Software Engineering* 36.2 (2010), pp. 226–247.

[51] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[52] J. Jiang, H. Xu, and Y. Zhou. "RULF: Rust Library Fuzzing via API Dependency Graph Traversal". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2021, pp. 581–592.

[53] M. Khari and P. Kumar. "An extensive evaluation of search-based software testing: a review". In: *Soft Computing* 23.6 (2019), pp. 1933–1946.

[54] J. C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394.

[55] B. Korel. "Automated software test data generation". In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879.

[56] B. Korel and A. Al-Yami. "Assertion-oriented automated test data generation". In: *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE Computer Society, 1996, pp. 71–80.

[57] B. Korel. "Dynamic method for software test data generation". In: *Software Testing, Verification and Reliability* 2.4 (1992), pp. 203–213.

[58] H. M. Le. "KLUZZER: Whitebox Fuzzing on Top of LLVM". In: *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2019, pp. 246–252.

[59] B. Li et al. "Many-Objective Evolutionary Algorithms: A Survey". In: *ACM Comput. Surv.* 48.1 (2015).

[60] G. Li, I. Ghosh, and S. P. Rajan. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs". In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 609–615.

[61] Y. Lin et al. “Graph-Based Seed Object Synthesis for Search-Based Unit Testing”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1068–1080.

[62] S. Lukasczyk, F. Kroiß, and G. Fraser. “Automated Unit Test Generation for Python”. In: *Search-Based Software Engineering*. Cham: Springer International Publishing, 2020, pp. 9–24.

[63] P. McMinn. “Search-Based Failure Discovery Using Testability Transformations to Generate Pseudo-Oracles”. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1689–1696.

[64] P. McMinn. “Search-based software test data generation: a survey”. In: *Software Testing, Verification and Reliability* 14.2 (May 2004), pp. 105–156.

[65] P. McMinn. “Search-Based Software Testing: Past, Present and Future”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society, 2011, pp. 153–163.

[66] M. Miller. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. URL: `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf` (visited on 06/28/2021).

[67] G. J. Myers, C. Sandler, and T. Badgett. “The art of software testing”. In: (2011).

[68] E. J. Njor and H. Gústafsson. “Static Taint Analysis in Rust”. In: (2021).

[69] C. Pacheco and M. D. Ernst. “Randoop: feedback-directed random testing for Java”. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 815–816.

[70] A. Panichella, J. Campos, and G. Fraser. “EvoSuite at the SBST 2020 Tool Competition”. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 549–552.

[71] A. Panichella, F. M. Kifetew, and P. Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: *IEEE Transactions on Software Engineering* 44.2 (2018), pp. 122–158.

[72] A. Panichella, F. M. Kifetew, and P. Tonella. "Reformulating Branch Coverage as a Many-Objective Optimization Problem". In: (2015).

[73] R. T. Prosser. "Applications of boolean matrices to the analysis of flow diagrams". In: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. 1959, pp. 133–138.

[74] H. Rocha et al. "Map2Check: Using Symbolic Execution and Fuzzing". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2020, pp. 403–407.

[75] J. M. Rojas et al. "A detailed investigation of the effectiveness of whole test suite generation". In: *Empirical Software Engineering* 22.2 (2017), pp. 852–893.

[76] J. M. Rojas et al. "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation". In: *Search-Based Software Engineering*. Cham: Springer International Publishing, 2015, pp. 93–108.

[77] D. Romano, M. Di Penta, and G. Antoniol. "An Approach for Search Based Testing of Null Pointer Exceptions". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, pp. 160–169.

[78] K. Sen and G. Agha. "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools". In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 419–423.

[79] N. Shamrell-Harrington. *Microsoft joins Rust Foundation*. URL: https://cloudblogs.microsoft.com/opensource/2021/02/08/microsoft-joins-rust-foundation/ (visited on 06/28/2021).

[80] S. Shamshiri et al. "Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?" In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1367–1374.

[81] Stack Overflow. *Stack Overflow Developer Survey 2020*. URL: https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved (visited on 05/02/2021).

[82] J. V. Stoep and S. Hines. *Rust in the Android platform*. URL: https://security.googleblog.com/2021/04/rust-in-android-platform.html (visited on 10/06/2021).

[83] A. Ter-Sarkisov and S. Marsland. "Convergence Properties of (μ + λ) Evolutionary Algorithms". In: vol. 25. 1. 2011.

[84] The Rust Core Team. *Announcing Rust 1.0*. URL: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html (visited on 06/28/2021).

[85] N. Tillmann and J. de Halleux. "Pex–White Box Test Generation for .NET". In: *Tests and Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 134–153.

[86] P. Tonella. "Evolutionary Testing of Classes". In: *SIGSOFT Softw. Eng. Notes* 29.4 (2004), pp. 119–128.

[87] S. Vogl et al. "EVOSUITE at the SBST 2021 Tool Competition". In: *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE Computer Society, 2021, pp. 28–29.

[88] L. D. Whitley. *The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best*. Citeseer, 1989.

[89] M. Zalewski. *American fuzzy lop*. 2014.