Generating Mutation Tests Using an Equivalence Prover

Christian Martin

August 9, 2022

Generating Mutation Tests Using an Equivalence Prover

Christian Martin

Karlsruher Institut für Technologie (KIT)

Abstract. Creating test cases and assessing their quality is a necessary but time-consuming part of software development. Mutation Testing facilitates this process by introducing small syntactic changes into the program and thereby creating mutants. The quality of test cases is then measured by their ability to differentiate between those mutants and the original program. Previous studies have also explored techniques to automatically generate test inputs from program mutants. However, the symbolic approaches to this problem do not handle unbounded loops within the program. In this paper, we present a method using the equivalence prover *LLRêve* [7] to generate test cases from program mutants. We implemented the method as a pipeline and evaluated it on example programs from the learning platform *c4learn* [19] as well as programs from the library *diet libc* [15]. Our results have shown that this method takes up to multiple hours to run on these programs, but the generated test cases are able to detect half of the known errors. As an extension to the method, we also present mutations which introduce an additional parameter into the program. We then show on an example that a finite set of test cases can be sufficient to kill the mutant for all values of the parameter.

Keywords: Mutation Testing · Equivalence Prover

1 Introduction

During development of a program, the program will at some point accumulate errors in its source code. Those errors lead to incorrect behaviour of the software under certain circumstances. To prevent incorrect behaviour, it is therefore desirable to find the errors in the source code. One established method of finding errors is executing the program on test inputs and checking the behaviour. Writing test cases however, which include test inputs and checks for the output, is a time-consuming matter. Ideally, we want to automatically generate a minimal set of test cases which is sufficient to find all errors within the program. Nevertheless, the problem of finding such a set is also non-trivial.

In this paper, we use a technique called program mutation. It was originally conceived by De Millo et al. in 1978 [5]. Program mutation is the act of generating a set of mutants from a program. Each mutant's source code differs from the original program's source code only by a "small amount". If the mutant behaves differently than the original, then there is an error in the original or in the mutant. A test case which is able to distinguish between the original and the mutant is therefore able to detect this error. In this case, the test case is said to *kill* the mutant.

The main contributions of this paper are a method for generating test cases from mutants using an equivalence prover as well as an implementation of this method which we also evaluated on a set of benchmark programs. Furthermore, with parametric mutants we present a novel type of mutants which each represent a family of mutants as well as a method to kill them.

To create test cases with our method, a set of mutants is generated from the original program. Then an equivalence prover checks whether the original and the mutant are equivalent. If they are equivalent, then no test case can be deduced from the mutant. Otherwise, the equivalence prover provides a counterexample which we can use as a test input. In this paper, we assume that a reference implementation exists, which defines the correct behaviour of the original program. We run the reference implementation on each test input to determine the expected output. In the absence of a reference implementation, the user has to provide checks for the test cases to determine whether the output is correct.

As a proof of concept, we implemented our approach as a pipeline and evaluated it on a number of example programs from *c4learn* [19] as well as some library functions from *diet libc* [15]. The evaluation shows the benefits of the approach as well as limitations due to the used tools or due to the method of program mutation itself.

Furthermore, we extended our approach by the concept of parametric mutants to use the symbolic nature of the equivalence prover. By introducing the parametric mutants which each contains an additional parameter, we are able to represent a family of mutants with one parametric mutant. Killing all mutants of one of these families is achieved by generating a set of test cases which kill the parametric mutant for all possible values of the parameter.

Section 2 serves as an introduction to the fundamentals of mutation testing as well as the workings of the equivalence prover *LLRêve* which we used in our work. In section 3, we present previous studies which explored ways to generate test cases from program mutants. Our own approach is explained further in section 4, where we also show the results of our evaluation. Section 5 illustrates our concept to use parametric mutants and which advantages it may have. Finally, we draw our conclusions in section 7.

2 Background

2.1 Mutation Analysis

Mutation analysis is a method introduced by De Millo et al. in 1978 [5] for assessing the quality of test cases. To achieve this, mutation analysis uses *first-order mutants* of the program under test. A first-order mutant is a program which differs from the program under test by only a small syntactic change in the source code. In a different paper by De Millo et al. from 1988 [4], they introduced mutation operators which specify the exact syntactic changes that are allowed to generate first-order mutants. Some of these syntactic changes introduce a semantic difference between the original program under test and the mutant. In this case, at least one of both programs contains an error. A test case which differentiates between the original and the mutant therefore is able to detect this error.

The method proposed by De Millo et al. in their 1978 paper starts by running the program under test on all given test cases. If the program fails at least one of the test cases, then an error has been found and can be fixed. If the program passes all of the test cases, then it either does not contain any errors or the test cases are insufficient to find one of the errors in the program. To find out which case applies, a set of first-order mutants is generated from the program under test. Then, each mutant is executed on all test cases. If a mutant returns a different result than the original program, the mutant is considered *dead*. On the other hand, if a mutant returns the same result as the original on each test case, it is considered *live*. A mutant can be live either because it is equivalent to the original or because it represents an error which the test cases cannot detect. For each live mutant, the programmer must determine the reason why the syntactic difference between the original and the mutant did not matter to the test results.

William E. Howden [12] extended De Millo et al.'s notion of *killing* mutants. If a mutant is killed by producing a different output than the original, the mutant is said to be *strongly killed*. If on the other hand the mutant and the original have differing program states at the same point during their execution, but do not necessarily produce different outputs, the mutant is said to be *weakly killed*. By this definition, all mutants which have been strongly killed also have been weakly killed. This extension of killing is used in most of the related work which focuses on generating test inputs that weakly kill mutants.

In this paper, we define that a set of test cases fulfills the *weak mutation coverage criterion* if and only if the test cases weakly kill all non-equivalent mutants. This is similar to the *weak mutation coverage criterion* used in a related paper by Papadakis et al. [18]. Analogously, we define the *strong mutation coverage criterion* to be fulfilled if all non-equivalent mutants have been strongly killed.

2.2 Equivalence Prover

In this project, we use the equivalence prover *LLRêve* [14]. An equivalence prover is a tool which, given two programs, proves or disproves their equivalence. A pair of programs is considered equivalent if and only if they produce identical outputs when executed on identical inputs.

In the case of LLRêve, the tool takes two program source codes written in C as input and creates an SMT file. If the SMT file is satisfiable, then this also means that the two programs are equivalent. All SMT clauses created by LLRêve are *Horn clauses*, meaning they each are an implication $B \to H$ from a body B

to a head H, where B is the conjunction of one or more subexpressions. If the head is equal to `false`, then the entire Horn clause is equal to $\neg B$.

For a pair of programs P_1 and P_2 to be compared by LLRêve, they both must have the same set of synchronization points such that each loop contains at least one synchronization point. For each of these synchronization points, LLRêve declares a coupling predicate placeholder C_i in the SMT file which represents a relation between the program states of both programs at this point. Whenever there exists a pair of paths in the two programs' control flows from a synchronization point i to a subsequent synchronization point j, the SMT file contains an assertion of the form

$$\forall x, y, x', y' : C_i(x, y) \wedge \pi_{ij}(x, x') \wedge \rho_{ij}(y, y') \rightarrow C_j(x', y')$$

where π_{ij} and ρ_{ij} define the state transitions from program states x and y of P_1 and P_2 at i to the program states x' and y' at j. Additionally, the coupling predicate C_{in} is included in another clause stating

$$\forall x, y : x = y \rightarrow C_{\text{in}}(x, y).$$

This means that the SMT clauses within the file apply to all cases in which the inputs of both programs are identical. On the other hand, there is also a clause which has the form

$$\forall x', y' : C_{\text{out}}(x', y') \rightarrow x' = y'$$

where C_{out} is the coupling predicate on the outputs of the programs. Therefore the SMT file states that any time both programs are executed on the same input, the output of both programs is also the same. In short, if the SMT file is satisfiable then the two programs are equivalent.

3 Related Work

Multiple approaches have already been proposed for generating test inputs from program mutants.

The first approach we are aware of was presented in 1991 by De Millo and Offutt [6] and uses *constraint based testing*. The authors form a constraint for strongly killing a mutant which is then solved via multiple heuristics. The solution for the constraint is a test input which strongly kills the mutant.

Two approaches rely on the usage of model checking. Ammann et al. [1] focus on the specifications of a software system. Instead of mutating the software itself, the specifications are mutated. The model checker is then used to scan the mutated specifications for any inconsistencies. If an inconsistency is found, the model checker yields the test input as well as the expected output of the software system.

Riener et al. [20] on the other hand first convert the program to a meta mutant and then to logical formulae before applying an SMT solver. The meta mutant is a program which contains all mutations. A global variable controls which of the mutants is run in the current execution. The SMT solver outputs a test input for each mutant until all the remaining mutants are equivalent.

Three papers were co-authored by Papadakis and Malevris. The 2010 paper [18] by Papadakis et al. reduces weak mutation coverage to branch coverage. This is achieved by converting the program under test to a meta program. The meta program is equivalent to the original program except for an additional branch for each mutation which is taken if and only if the respective mutant is weakly killed. Thus a set of test cases for the meta program which meet the branch coverage criterion also meet the weak mutation coverage criterion on the original program.

Papadakis and Malevris presented another approach in 2012 [16] based on symbolic execution. They convert the program under test to an extended control flow graph which is similar to the meta program from their 2010 approach. They search the "shortest" path in the graph from the input vertex to each mutated vertex. Then the path constraints for each of these paths are solved to get the respective test input which weakly kills the mutant.

In 2013, Papadakis and Malevris pursued a different approach [17]. This approach also requires converting the program under test to a meta program, but continues by using *hill climbing*. This means that at each of multiple steps, a single program argument is slightly varied to measure the influence of the change to a fitness function. The fitness function depends on the distance of the executed control flow to the mutated statement and, if the mutated statement is reached, on the influence of the statement on the next program state and the output.

Another two approaches are based on *concolic execution*. Concolic execution is a hybrid of a concrete execution and a symbolic execution. The concrete execution is started on random inputs while collecting symbolic path constraints along the taken path. To get a new path, a single branch constraint is inverted. If this modification does not create a contradiction, then the constraints are solved to get a concrete input for the new path.

One approach using concolic execution was proposed by Zhang et al. in 2010 [22]. Their approach is similar to the one by Papadakis et al. in that they convert the program under test to a meta program and then generate test inputs with branch coverage on the meta program. Zhang et al. use an existing concolic-execution-based branch coverage tool to generate these test inputs.

Harman et al. [10] base their approach on the work by Zhang et al. The authors focus on strongly killing higher-order mutants. Contrary to first-order mutants, higher-order mutants are created by mutating the original program multiple times for each mutant. In the case of Harman et al.'s work, they combine mutants to create new mutants of increasing order. Using search based software testing, they find paths passing all mutated statements for each mutant if possible.

The approach by Fraser and Zeller [8] uses an *evolutionary algorithm*. The test cases considered by the approach are encoded genetically. Then crossovers and genetic mutations can be applied to create new test cases. A fitness function then filters the test cases based on the distance of the control flow to the mutated statement and based on the influence of the mutated statement on the output.

An approach from 2020 was presented by Baer et al. [2]. The authors use symbolic execution to extract symbolic program states before and after the mutated statement. An equivalence checker constructed by the authors then generates a formula from the program states. Solving multiple formulae at once yields a test input which weakly kills multiple mutants.

4 Pipeline for Killing Mutants

4.1 Approach

In our approach, we first generate a set of mutants for the original program under test. Since for each non-equivalent mutant an error exists in the original or in the mutant, a test case which distinguishes between the two programs must be able to detect this error. Therefore we want to generate test cases with strong mutation coverage.

For this purpose, we use an equivalence prover to compare each mutant with the original. If the mutant is equivalent, we cannot infer any information about any errors within the programs. If the mutant is not equivalent, then the equivalence prover generates a counterexample. The counterexample includes an input for which the mutant and the original produce different outputs. This means that we can extract a test input from the counterexample which strongly kills the mutant.

To create a full test case, we add the expected output of the program according to its intended behaviour. In our case, we assume that a reference implementation exists which complies with the intended behaviour. In practice, the reference implementation may be a simpler yet slower program for the same purpose. We obtain the expected output for the test case by executing the reference implementation on the test input and using its output.

4.2 Implementation

We implemented our approach as a pipeline which is outlined in figure 1. The pipeline takes the source code of the original program written in C as well as the executable of the reference implementation as input. Generating the mutants in the first step is done via one of three existing mutant generators: *Dextool Mutate* [3], *MUSIC* [13] or *Universal Mutator* [9]. The pipeline can also be adapted to use any mutant generator which mutates C source code.

As the equivalence prover of the pipeline, we use *LLRêve* [7] (see section 2.2). LLRêve creates an SMT file such that if the SMT file is satisfiable, then the two programs are equivalent. There is also a possibility of the SMT file not being satisfiable while the two programs are equivalent, but this only may lead to more test cases than necessary and does not affect the strong mutation coverage criterion.

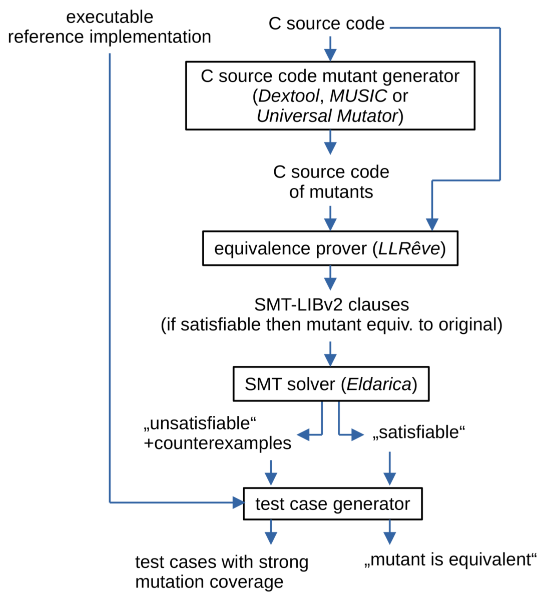

Fig. 1. Workflow of the pipeline

The pipeline uses the SMT solver *Eldarica* [11] to check the satisfiability of the SMT file. If the SMT file is not satisfiable, Eldarica returns a counterexample. The counterexample consists of concrete invocations of the predicates in the file which form a contradiction. The arguments for the initial predicate are two identical inputs to the programs which produce different outputs. These inputs are the test input which will be used for the test case.

After extracting the test input, the reference implementation is executed on the input and produces the expected output. Finally, a test case is generated for the C unit testing framework *Unity* [21] which executes the original program on the test input and checks whether the output equals the expected output. Unity can also be interchanged for any unit testing framework for C through small modifications of the pipeline.

Avoiding redundant executions of LLRêve and Eldarica During first experiments with the pipeline, we noticed that frequently, Eldarica generates the same counterexample for multiple mutants. Therefore we used this opportunity to reduce the number of calls to LLRêve and Eldarica.

After each new counterexample found by Eldarica, the pipeline uses all counterexamples to generate test cases. These test cases, however, are not created by using the reference implementation, but instead get their expected outputs from an executable of the original program under test. For this reason, a mutant fails a test case if and only if it is killed by this test case. This provides us a method to check which of the subsequent mutants are already killed by the previous test cases. For these mutants we do not need additional test cases and therefore we can omit to execute LLRêve and Eldarica on these mutants.

Note that at the end, the pipeline has to generate test cases once again, but this time by using the reference implementation to determine the expected outputs. These are the actual test cases returned by the pipeline.

Handling missing loops Since LLRêve synchronizes the execution of loops between both programs, mutants are required to have the same number of loops as the original. However, this is not the case if the respective mutation operator deleted the loop. LLRêve's optimization also removes a loop, if the program always terminates before the loop or if the loop condition is mutated to be a constant 0. If a loop is missing in the mutant, LLRêve generates an error message and aborts. In order to still be able to kill these kinds of mutants, we modified all three mutant generators to generate additional mutants.

The first type of mutant deals with the issue that the condition of an `if` statement may be mutated to be a constant 0 or a constant 1. This way, a loop within one of the branches may never be executed and therefore may be removed by LLRêve's optimization. The mutation operator replaces the condition of an `if` statement with an undefined external function. LLRêve's optimization can neither assume that the condition always holds nor that the condition never holds. However, when Eldarica tests the SMT clauses for satisfiability, it checks whether the clauses are satisfiable for all implementations of the external function. Therefore, Eldarica outputs a counterexample, if such a counterexample exists for at least one implementation of the external function. In this case, it always is a counterexample for a mutant with a constant mutated `if` condition.

The second type of mutant mimics the cases in which the loop is removed or has no iterations. This mutant also uses an undefined external function. The function sets an auxiliary variable which is then used as the condition of an empty loop. This empty loop also cannot be removed by LLRêve's optimization. If Eldarica finds a counterexample, it must be a counterexample for the mutant where the auxiliary variable is set to 0 since all other cases would lead to an infinite loop.

4.3 Evaluation Setup

To evaluate our pipeline, we used 28 benchmark programs. 18 of these programs are example programs from *c4learn* [19] the other 10 programs are single functions from *diet libc* [15]. *c4learn* is a website for learning how to write programs in C. *diet libc* is a C standard library optimized to have a small size.

Our choice of benchmark programs was limited in part due to the tools we used for our pipeline. LLRêve does not operate on any programs which contain bitwise operators applied to integers, programs with function pointers which were only initialized at runtime or programs which contain infinite loops. Moreover, Eldarica can analyze SMT files which either contain multiplication operations or arrays, but no SMT files which contain both.

Each of the programs we used contained one error. In each of the c4learn example programs, we included an artificial error. For each of the diet libc functions, we used an older version which had an error that was fixed in a later version with a small syntactic change. When running the pipeline on the benchmark programs, we tested which of these errors the generated test cases were able to find. Additionally, we counted how many mutants were generated, how many were killed and how many were equivalent. We also measured the time the pipeline took to generate the mutants, the time for generating the counterexamples and the time for generating the test cases.

When analyzing each mutant, in most cases, Eldarica needed less than 60 seconds. On the other hand, in some cases, if it did not finish within 60 seconds for a single mutant, it also did not finish after one hour. For this reason, we introduced a limit to the execution time of Eldarica of 60 seconds per mutant.

4.4 Results

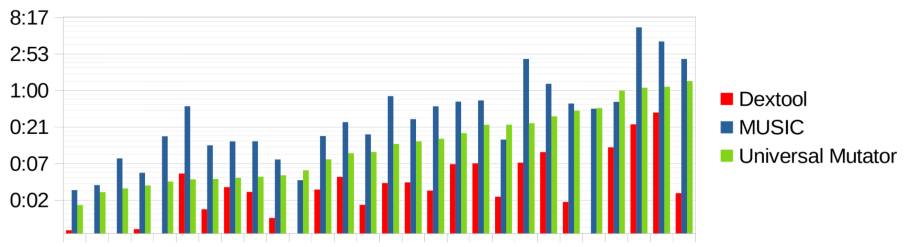

Fig. 2. Logarithmic chart of the runtime of the pipeline in hours:minutes using each of the three mutant generators on each benchmark program

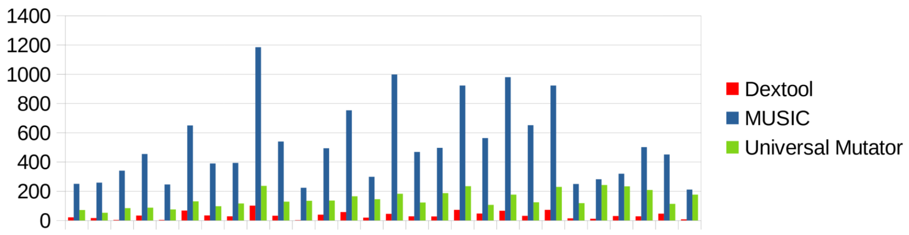

Fig. 3. Number of mutants generated by each of the three mutant generators from each benchmark program

All results of the evaluation are listed in the appendix in section A. The results show that the total runtime of the pipeline ranges from a single-digit

	Dextool Mutate	MUSIC	Universal Mutator
Coefficient for total number of mutants	0.20	0.01	0.43
Coefficient for timed-out mutants	0.96	0.97	0.96

Table 1. Correlation coefficients between number of mutants and time for analysis with LLRêve and Eldarica, for each used mutant generator

amount of minutes to multiple hours, as can also be seen in figure 2. It also shows that the runtime of the pipeline has some correlation between the different mutant generators, with all pairwise correlation coefficients ranging from 0.47 to 0.90. The runtime on the other hand is only weakly correlated to the number of generated mutants with correlation coefficients between 0.01 and 0.43, as can be seen in the differences between the figures 2 and 3.

The time the pipeline needs for generating counterexamples for the programs is highly correlated to the number of mutants on which Eldarica exceeded the 60 second time limit, as is shown in table 1. It is also apparent that the programs, for which the pipeline needs substantially more runtime, are not necessarily more complex than the programs with low pipeline runtimes. There is however a clear difference between the three mutant generators. Out of 28 errors, the pipeline found 14 using Dextool Mutate, 16 using Universal Mutator and 18 using MUSIC. For comparison, MUSIC generates on average 3.7 times as many mutants as Universal Mutator, which in turn generates on average 9.4 times as many mutants as Dextool Mutate. The total runtime of the pipeline when using MUSIC is on average 2.6 times as long as the runtime when using Universal Mutator, which in turn is on average 6.3 times as long as the runtime when using Dextool Mutate.

4.5 Example

One of the benchmark programs we used is `isxdigit`. `isxdigit` is a function of diet libc which determines whether the character encoded in the integer parameter `c` is a hexadecimal digit. Listing 1.1 shows the source code of version 1.2. Version 1.2 of `isxdigit` had the error that the last parenthesized term contained a logical OR. As Version 1.3 in listing 1.2 shows, the correct operator was a logical AND. This had the consequence that version 1.2 returned `1` on every possible input.

In our evaluation we used version 1.2 as the program under test and version 1.3 as the reference implementation. We noticed that all mutant generators generated a high fraction of equivalent mutants compared to the other benchmark programs. This was the case due to the mutants also always returning `1` as long as the last term or the logical OR before the last term were not mutated. The generated test cases found the error regardless of which mutant generator the pipeline used. All three mutant generators have a mutant operator which replaces the logical OR with a logical AND. As a consequence, all three mutant

```
int __isxdigit_ascii(int c) {
  return ((c>='0'&&c<='9') || (c>='A'&&c<='F') ||
          (c>='a'||c<='f'));
}
```

Listing 1.1. Source code for `isxdigit` version 1.2

```
int __isxdigit_ascii(int c) {
  return ((c>='0'&&c<='9') || (c>='A'&&c<='F') ||
          (c>='a'&&c<='f'));
}
```

Listing 1.2. Source code for `isxdigit` version 1.3

generators created a mutant which was identical to version 1.3. In this case, the mutant was “more correct” than the original program under test.

5 Parametric Mutants

5.1 Idea

Some usual mutant operators replace a constant or a variable with a specific integer literal. In case of the mutant generators we used, the literal which replaces the previous expression is one of `0`, `1`, `-1`, `MIN_INT` and `MAX_INT`. There is however no guarantee that these five values are sufficient to represent all errors which have to be fixed by replacing a constant or variable with another constant.

We propose a new kind of mutant: Instead of replacing some expression with a literal, we create a mutant by replacing the expression with an additional parameter. Hence we call these mutants *parametric mutants*. When applying LLRêve and Eldarica, Eldarica generates a counterexample with a specific value for the parameter. This is equivalent to killing a mutant which has the specific value as a literal in place of the parameter. As we will show, there are parametric mutants which can be killed for all possible values of the parameter, for which the mutant is not equivalent, with only a small set of test cases. This can be thought of as killing infinitely many non-parametric mutants with finitely many test cases.

5.2 Approach

Our goal is to *kill all non-equivalent parameter values* of a parametric mutant. By *killing a parameter value* of a parametric mutant, we mean killing the parametric mutant for that specific parameter value. We call a parameter value *equivalent* if and only if the parametric mutant is equivalent to the original program under test for that parameter value. To kill all non-equivalent parameter values, we execute Eldarica multiple times and generate multiple test cases from one parametric mutant. We require that after each execution of Eldarica, the generated test cases will be able to kill more parameter values than before that execution. When Eldarica no longer finds a counterexample which increases the number of killed parameter values, we are done.

In section 2.2, we described how LLRêve generates an SMT file for a mutant such that the mutant is equivalent if the SMT file is satisfiable. We use this file to create a new SMT file for a parametric mutant. Before each execution of Eldarica, we already found $n \in \mathbb{N}_0$ test inputs $t_1, ..., t_n$. Accordingly, in the first iteration, n is equal to 0. The file is constructed in each iteration such that if the file is satisfiable, all parameter values which are not killed by $t_1, ..., t_n$ are equivalent.

We treat the parameter p of the parametric mutant like another input which remains unchanged throughout the program. This means each assertion which describes parts of the control flows of both programs now has the following form:

$$\forall x, y, x', y', p : C_i(x, y, p) \land o_{ij}(x, x') \land \mu_{ij}(y, y', p) \to C_j(x', y', p)$$

Here o_{ij} describes the program state transition for the original program under test while μ_{ij} describes the program state transition for the parametric mutant. After adding p to the coupling predicates, we duplicate each coupling predicate C_i n times to get the predicates $C_i^{(1)}, ..., C_i^{(n)}$. Additionally, we duplicate each assertion for the control flows n times while using the k-th duplicated predicates $C_i^{(k)}$ for the k-th duplicate of the assertion.

Essentially, we now have clauses which describe the control flows of the original and the parametric mutant $n+1$ times. We also still have a slight modification of the input clause:

$$\forall x, y, p : x = y \to C_{\text{in}}(x, y, p)$$

Now we define a new input clause for each duplicated input predicate:

$$\forall p : C_{\text{in}}^{(k)}(t_k, t_k, p)$$

This means that the k-th test input is a valid input for the k-th duplicated control flows of the original and the parametric mutant. As a result, we now have clauses which define one execution of the original and the mutant on any arbitrary input as well as n executions on the previous test inputs. To join all executions, we replace the output clause with a new clause:

$$\forall ... : C_{\text{out}}(x', y', p) \ \land \ \bigwedge_{k=1}^{n} C_{\text{out}}^{(k)}(x'_k, y'_k, p) \ \land \ \bigwedge_{k=1}^{n} x'_k = y'_k \ \to \ x' = y'$$

The universal quantifier refers to all outputs $x', y', x'_1, y'_1, ..., x'_n, y'_n$ as well as the parameter p. With this last clause we created an SMT file still only consisting of horn clauses which states that for any parameter value p, if the outputs of the original and the mutants are identical on each of the previous test inputs, then the outputs of the original and the mutant are identical on any input. In

```
int add_one(int x) {
    return x + 1;
}
```

Listing 1.3. Original of the first example

```
int add_one(int x) {
    return x + p;
}
```

Listing 1.4. Parametric mutant of the first example

other words, if the previous test inputs are not able to kill the parameter value p, then no test input exists which kills p.

If Eldarica finds a counterexample for this SMT file, then the counterexample includes a new test input such that the previous test inputs are not able to kill p, but the new test input is. Including the new test input into the set of test inputs therefore increases the number of parameter values killed by the set.

5.3 Examples

The amount of test cases needed to kill all non-equivalent parameter values can be finite or infinite depending on the specific original and the mutant. This is demonstrated via the following three examples.

The first example seen in listings 1.3 and 1.4 adds one to the input `x` and then returns the result. Any parameter value `p` not equal to `1` will always produce a different result for the mutant than the original. This means that one test input is sufficient to kill all parameter values except for the value `1`.

The second example in listings 1.5 and 1.6 checks whether the input `x` is greater than zero and then outputs `1` or `0` accordingly. If `p` is greater than zero, any value of `x` less or equal to zero will produce the same results for the original and the mutant. If `p` is less than zero, any value of `x` greater than zero will produce the same results for the original and the mutant. Thus, to kill all parameter values except for `0`, we need two test inputs. Input `x = 0` kills all parameter values less than zero, while `x = 1` kills all parameter values greater than zero.

Unfortunately, there also exist examples like the one in listings 1.7 and 1.8. The original program returns whether the input `x` is equal to zero. The parametric mutant however returns whether `x` is equal to zero or equal to `p`. This means that the parametric mutant only behaves differently to the original if `x` equals `p`. To kill all parameter values except `0`, one has to create a new test input for each parameter value, which is the worst case scenario.

```
int is_positive(int x) {
    if (x > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Listing 1.5. Original of the second example

```
int is_positive(int x) {
    if (x > p) {
        return 1;
    } else {
        return 0;
    }
}
```

Listing 1.6. Parametric mutant of the second example

```
int is_zero(int x) {
    if (x == 0 || x == 0) {
        return 1;
    } else {
        return 0;
    }
}
```

Listing 1.7. Original of the third example

```
int is_zero(int x) {
    if (x == 0 || x == p) {
        return 1;
    } else {
        return 0;
    }
}
```

Listing 1.8. Parametric mutant of the third example

6 Discussion

As seen in section 4.4, the pipeline in its current state takes multiple hours on some small programs, which is infeasible in real applications. Additionally, the tools within the pipeline limit for which programs test cases can be generated by the pipeline.

These issues however are due to the specific tools used and not a consequence of the underlying concept. Program mutation itself on the other hand also has a flaw which became apparent during evaluation. Most of the errors from diet libc, which the generated test cases could not find, were fixed in later versions by adding a check for a special case. Adding a check for special case is not a syntactic modification that program mutation can mimic since there are too many possibilities for which special case to check and how to handle it.

The choice of the mutant generator proved important both for the total runtime of the pipeline and the number of errors the generated test cases were able to find. The difference in runtime however was significantly greater than the difference in the number of found errors. This indicates that generating more mutants is not necessarily worth the effort. It seems to be more feasible to only attempt to generate mutants that are killed by test cases which detect as many errors as possible.

7 Conclusion

In summary, we presented a method to generate test cases from mutants using an equivalence prover. We implemented this method as a pipeline using multiple existing tools. The pipeline was slow and had limitations, but was able to find at least half of the errors during the evaluation. In addition to that, we introduced the concept of parametric mutants. Parametric mutants represented an entire family of mutants. Killing all parameter values of the parametric mutant could in some cases be achieved with a finite number of test cases, but in other cases the number of test cases was linear in the number of possible parameter values.

As future work, one may exchange the mutant generator, the equivalence prover or the SMT solver for other tools to reduce the high runtime for generating counterexamples on some mutants or to remedy the limitations on the programs the pipeline can be applied to. Furthermore, the approach for killing parametric mutants can be implemented into the pipeline. It may also be useful to investigate which kinds of parametric mutants only need a finite number of test cases to be killed for all parameter values.

References

1. Ammann, P., Black, P., Majurski, W.: Using model checking to generate tests from specifications. In: Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241). pp. 46–54 (Dec 1998). https://doi.org/10.1109/ICFEM.1998.730569

2. Baer, M., Oster, N., Philippsen, M.: MutantDistiller: Using Symbolic Execution for Automatic Detection of Equivalent Mutants and Generation of Mutant Killing Tests. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 294–303 (Oct 2020). https://doi.org/10.1109/ICSTW50294.2020.00055
3. Brännström, J.: Dextool Mutate (Oct 2021), https://github.com/joakim-brannstrom/dextool/tree/master/plugin/mutate, original-date: 2015-10-06T06:41:19Z
4. DeMillo, R., Guindi, D., McCracken, W., Offutt, A., King, K.: An extended overview of the Mothra software testing environment. In: [1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis. pp. 142–151 (Jul 1988). https://doi.org/10.1109/WST.1988.5369
5. DeMillo, R., Lipton, R., Sayward, F.: Hints on Test Data Selection: Help for the Practicing Programmer. Computer **11**(4), 34–41 (Apr 1978). https://doi.org/10.1109/C-M.1978.218136, conference Name: Computer
6. DeMillo, R., Offutt, A.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering **17**(9), 900–910 (Sep 1991). https://doi.org/10.1109/32.92910, conference Name: IEEE Transactions on Software Engineering
7. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. pp. 349–360. ASE '14, Association for Computing Machinery, New York, NY, USA (Sep 2014). https://doi.org/10.1145/2642937.2642987, https://doi.org/10.1145/2642937.2642987
8. Fraser, G., Zeller, A.: Mutation-Driven Generation of Unit Tests and Oracles. IEEE Transactions on Software Engineering **38**(2), 278–292 (Mar 2012). https://doi.org/10.1109/TSE.2011.93, conference Name: IEEE Transactions on Software Engineering
9. Groce, A., Holmes, J., Marinov, D., Shi, A., Zhang, L.: An extensible, regular-expression-based tool for multi-language mutant generation. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 25–28. ACM, Gothenburg Sweden (May 2018). https://doi.org/10.1145/3183440.3183485, https://dl.acm.org/doi/10.1145/3183440.3183485
10. Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11. p. 212. ACM Press, Szeged, Hungary (2011). https://doi.org/10.1145/2025113.2025144, http://dl.acm.org/citation.cfm?doid=2025113.2025144
11. Hojjat, H., Rümmer, P.: The ELDARICA Horn Solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–7 (Oct 2018). https://doi.org/10.23919/FMCAD.2018.8603013
12. Howden, W.: Weak Mutation Testing and Completeness of Test Sets. IEEE Transactions on Software Engineering **SE-8**(4), 371–379 (Jul 1982). https://doi.org/10.1109/TSE.1982.235571, conference Name: IEEE Transactions on Software Engineering
13. swtv kaist: MUtation analySIs tool with high Configurability and extensibility MUSIC (Sep 2021), https://github.com/swtv-kaist/MUSIC, original-date: 2017-07-11T07:21:16Z

14. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational Program Reasoning Using Compiler IR. In: Blazy, S., Chechik, M. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 149–165. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_12
15. von Leitner, F.: diet libc - a libc optimized for small size, https://www.fefe.de/dietlibc/
16. Papadakis, M., Malevris, N.: Mutation based test case generation via a path selection strategy. Information and Software Technology **54**(9), 915–932 (Sep 2012). https://doi.org/10.1016/j.infsof.2012.02.004, https://www.sciencedirect.com/science/article/pii/S095058491200047X
17. Papadakis, M., Malevris, N.: Searching and generating test inputs for mutation testing. SpringerPlus **2**(1), 121 (Mar 2013). https://doi.org/10.1186/2193-1801-2-121, https://doi.org/10.1186/2193-1801-2-121
18. Papadakis, M., Malevris, N., Kallia, M.: Towards automating the generation of mutation tests. In: Proceedings of the 5th Workshop on Automation of Software Test - AST '10. pp. 111–118. ACM Press, Cape Town, South Africa (2010). https://doi.org/10.1145/1808266.1808283, http://portal.acm.org/citation.cfm?doid=1808266.1808283
19. Pritesh: Learn Programming Tutorials Step By Step - c4learn.com, https://www.c4learn.com/
20. Riener, H., Bloem, R., Fey, G.: Test Case Generation from Mutants Using Model Checking Techniques. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 388–397 (Mar 2011). https://doi.org/10.1109/ICSTW.2011.55
21. VanderVoord, M., Karlesky, M., Williams, G.: Unity – Unit Testing for C, http://www.throwtheswitch.org/unity
22. Zhang, L., Xie, T., Zhang, L., Tillmann, N., de Halleux, J., Mei, H.: Test generation via Dynamic Symbolic Execution for mutation testing. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–10 (Sep 2010). https://doi.org/10.1109/ICSM.2010.5609672, iSSN: 1063-6773

A Pipeline Evaluation Results

This section includes the results of the evaluation of our pipeline described in section 4. Each table includes the results for the 10 diet libc programs and the 18 c4learn programs while using the mutant generators Dextool Mutate, MUSIC and Universal Mutator. Tables 2 to 11 refer to diet libc programs while tables 12 to 29 refer to c4learn programs.

The total number of mutants is divided into mutants already killed by previously generated test cases, mutants which are equivalent to the original program under test, mutants which are each killed by a new counterexample found by Eldarica, mutants for which Eldarica had a runtime longer than 60 seconds and mutants which were either syntactically incorrect or contained features not supported by LLRêve or Eldarica.

The runtime of the pipeline is divided into three phases of the pipeline. In the first phase, a set of mutants is generated for the original program under test. In the second phase, the mutants are either checked for equivalence with the original or killed by previously generated test cases. In the third phase, the final set of test cases is generated and executed on the original. Each runtime is given in the format hours:minutes:seconds.

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	72	922	234
Mutants killed by previous test cases	60	770	149
Equivalent mutants	3	74	20
Mutants killed by new counterexample	9	42	17
Mutants with Eldarica timeout	0	3	3
Invalid mutants	0	33	45
Number of test cases	8	14	13
Time for generating mutants	0:01:12	0:00:05	0:00:03
Time for analyzing mutants	0:06:05	0:44:01	0:17:49
Time for generating final test cases	0:00:01	0:00:01	0:00:01
Error found by test cases?	no	no	no

Table 2. Results for `atoi` v.1.1 which parses a string as an integer

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	72	922	229
Mutants killed by previous test cases	56	768	149
Equivalent mutants	3	74	17
Mutants killed by new counterexample	9	41	17
Mutants with Eldarica timeout	2	6	3
Invalid mutants	0	33	43
Number of test cases	8	13	13
Time for generating mutants	0:01:39	0:00:07	0:00:07
Time for analyzing mutants	0:08:40	1:13:32	0:28:45
Time for generating final test cases	0:00:03	0:00:05	0:00:04
Error found by test cases?	no	no	no

Table 3. Results for `atol` v.1.1 which parses a string as a long integer

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	19	298	145
Mutants killed by previous test cases	8	169	35
Equivalent mutants	7	117	33
Mutants killed by new counterexample	4	6	5
Mutants with Eldarica timeout	0	0	0
Invalid mutants	0	6	72
Number of test cases	4	5	4
Time for generating mutants	0:00:31	0:00:04	0:00:03
Time for analyzing mutants	0:01:43	0:17:07	0:10:20
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 4. Results for `btowc` v.1.1 which converts a byte to a wide character

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	67	649	130
Mutants killed by previous test cases	10	94	72
Equivalent mutants	55	484	47
Mutants killed by new counterexample	2	10	2
Mutants with Eldarica timeout	0	0	0
Invalid mutants	0	61	9
Number of test cases	2	8	2
Time for generating mutants	0:01:13	0:00:04	0:00:02
Time for analyzing mutants	0:04:22	0:38:35	0:04:41
Time for generating final test cases	0:00:01	0:00:01	0:00:01
Error found by test cases?	yes	yes	yes

Table 5. Results for `isxdigit` v.1.2 which checks whether a character is a hexadecimal digit

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	2	223	134
Mutants killed by previous test cases	0	197	47
Equivalent mutants	0	8	15
Mutants killed by new counterexample	2	10	7
Mutants with Eldarica timeout	0	0	0
Invalid mutants	0	8	65
Number of test cases	2	7	5
Time for generating mutants	0:00:09	0:00:02	0:00:02
Time for analyzing mutants	0:00:14	0:04:33	0:06:04
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 6. Results for `memchr` v.1.2 which searches for a character within a memory area

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	14	249	118
Mutants killed by previous test cases	7	194	22
Equivalent mutants	0	0	0
Mutants killed by new counterexample	4	12	4
Mutants with Eldarica timeout	0	31	28
Invalid mutants	3	12	64
Number of test cases	3	4	4
Time for generating mutants	0:00:23	0:00:02	0:00:02
Time for analyzing mutants	0:02:03	0:41:46	0:33:58
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 7. Results for `stpncpy` v.1.1 which copies part of a string to a new location and returns a pointer to the end of the copy

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	30	319	233
Mutants killed by previous test cases	14	248	54
Equivalent mutants	0	1	2
Mutants killed by new counterexample	7	26	13
Mutants with Eldarica timeout	6	20	45
Invalid mutants	3	24	119
Number of test cases	5	10	10
Time for generating mutants	0:00:38	0:00:03	0:00:03
Time for analyzing mutants	0:11:11	0:43:45	1:00:47
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 8. Results for `strncmp` v.1.5 which compares parts two strings with respect to their lexicographic order

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	7	211	176
Mutants killed by previous test cases	0	108	11
Equivalent mutants	1	6	1
Mutants killed by new counterexample	5	15	8
Mutants with Eldarica timeout	1	61	68
Invalid mutants	0	21	88
Number of test cases	3	8	4
Time for generating mutants	0:00:13	0:00:02	0:00:03
Time for analyzing mutants	0:02:56	2:30:32	1:19:36
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 9. Results for `strncpy` v.1.8 which copies part of a string to a new location and returns a pointer to the start of the copy

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	11	281	242
Mutants killed by previous test cases	7	187	80
Equivalent mutants	0	0	0
Mutants killed by new counterexample	3	34	16
Mutants with Eldarica timeout	0	21	23
Invalid mutants	1	39	123
Number of test cases	2	12	11
Time for generating mutants	0:00:17	0:00:02	0:00:03
Time for analyzing mutants	0:00:40	0:36:01	0:36:41
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 10. Results for `strsep` v.1.1 which splits a string at a delimiter

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	28	501	208
Mutants killed by previous test cases	0	0	0
Equivalent mutants	0	3	0
Mutants killed by new counterexample	0	0	0
Mutants with Eldarica timeout	21	306	56
Invalid mutants	7	192	152
Number of test cases	0	0	0
Time for generating mutants	0:00:39	0:00:04	0:00:03
Time for analyzing mutants	0:22:19	6:13:19	1:05:49
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 11. Results for `strtok_r` v.1.1 which splits a string into tokens

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	57	753	165
Mutants killed by previous test cases	25	598	87
Equivalent mutants	8	66	32
Mutants killed by new counterexample	16	47	18
Mutants with Eldarica timeout	0	0	0
Invalid mutants	8	42	28
Number of test cases	11	20	11
Time for generating mutants	0:01:00	0:00:04	0:00:02
Time for analyzing mutants	0:04:03	0:24:21	0:09:58
Time for generating final test cases	0:00:02	0:00:03	0:00:02
Error found by test cases?	yes	yes	yes

Table 12. Results for "Program for deletion of an element from the specified location from Array" with an error that the wrong element is deleted

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	3	246	75
Mutants killed by previous test cases	0	163	40
Equivalent mutants	0	34	12
Mutants killed by new counterexample	2	15	5
Mutants with Eldarica timeout	0	2	0
Invalid mutants	1	32	18
Number of test cases	2	6	3
Time for generating mutants	0:00:10	0:00:02	0:00:01
Time for analyzing mutants	0:00:15	0:16:15	0:04:24
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	yes	no

Table 13. Results for the program "Find Factorial of Number without using function" with an error that the factorial of 1 is calculated incorrectly

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	21	250	71
Mutants killed by previous test cases	15	235	46
Equivalent mutants	1	0	0
Mutants killed by new counterexample	2	3	3
Mutants with Eldarica timeout	0	0	0
Invalid mutants	3	12	22
Number of test cases	2	3	3
Time for generating mutants	0:00:25	0:00:02	0:00:01
Time for analyzing mutants	0:00:39	0:03:25	0:02:13
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	yes	no

Table 14. Results for the program "Find Factorial of Number Using Recursion" with an error that the factorial of 1 is calculated incorrectly

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	66	979	176
Mutants killed by previous test cases	44	854	107
Equivalent mutants	15	60	30
Mutants killed by new counterexample	7	46	7
Mutants with Eldarica timeout	0	0	0
Invalid mutants	0	19	32
Number of test cases	6	23	5
Time for generating mutants	0:01:06	0:00:05	0:00:02
Time for analyzing mutants	0:01:45	0:14:43	0:22:41
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 15. Results for the program "Find greatest in 3 numbers" with an error that ties are not considered

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	33	454	87
Mutants killed by previous test cases	30	421	52
Equivalent mutants	0	4	14
Mutants killed by new counterexample	2	4	3
Mutants with Eldarica timeout	0	0	1
Invalid mutants	1	25	17
Number of test cases	2	4	3
Time for generating mutants	0:00:39	0:00:03	0:00:01
Time for analyzing mutants	0:00:27	0:05:37	0:03:54
Time for generating final test cases	0:00:01	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 16. Results for the program "Calculate gross salary of a person" with a rounding error

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	45	998	182
Mutants killed by previous test cases	21	749	81
Equivalent mutants	2	64	24
Mutants killed by new counterexample	16	117	34
Mutants with Eldarica timeout	0	1	0
Invalid mutants	6	67	43
Number of test cases	11	26	11
Time for generating mutants	0:00:52	0:00:05	0:00:02
Time for analyzing mutants	0:03:21	0:51:39	0:13:01
Time for generating final test cases	0:00:02	0:00:03	0:00:02
Error found by test cases?	no	no	no

Table 17. Results for "Program Insert element in an Array" with an error that the element in front of the inserted element is overwritten

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	47	563	106
Mutants killed by previous test cases	34	487	64
Equivalent mutants	2	15	6
Mutants killed by new counterexample	3	7	5
Mutants with Eldarica timeout	5	31	18
Invalid mutants	3	23	13
Number of test cases	3	4	3
Time for generating mutants	0:00:48	0:00:04	0:00:02
Time for analyzing mutants	0:06:40	0:45:35	0:22:37
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 18. Results for the program "Check for Armstrong Number in C" with an error that the loop in the program terminates too early if the last digit is a 1

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	46	451	113
Mutants killed by previous test cases	5	138	13
Equivalent mutants	2	7	6
Mutants killed by new counterexample	2	6	1
Mutants with Eldarica timeout	25	220	58
Invalid mutants	12	80	35
Number of test cases	2	5	1
Time for generating mutants	0:01:13	0:00:03	0:00:03
Time for analyzing mutants	0:31:00	4:08:01	1:07:36
Time for generating final test cases	0:00:03	0:00:03	0:00:03
Error found by test cases?	no	no	no

Table 19. Results for the program "Check Whether Number is Perfect Or Not" with an error that the input divided by two is not considered

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	28	393	116
Mutants killed by previous test cases	18	347	76
Equivalent mutants	0	0	0
Mutants killed by new counterexample	2	4	4
Mutants with Eldarica timeout	2	5	0
Invalid mutants	6	37	36
Number of test cases	2	4	4
Time for generating mutants	0:00:34	0:00:03	0:00:02
Time for analyzing mutants	0:03:11	0:14:01	0:04:52
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	yes	yes

Table 20. Results for the program "Check Whether Number is Prime or not" with an error that the square root of the input is not considered

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	101	1184	236
Mutants killed by previous test cases	87	1105	173
Equivalent mutants	6	53	26
Mutants killed by new counterexample	8	11	10
Mutants with Eldarica timeout	0	0	0
Invalid mutants	0	15	27
Number of test cases	8	10	10
Time for generating mutants	0:01:36	0:00:06	0:00:03
Time for analyzing mutants	0:01:40	0:13:59	0:05:01
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 21. Results for the program "Reads customer number and power consumed and prints amount to be paid" with an error that one of the edge cases yields a result of 0

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	34	389	97
Mutants killed by previous test cases	25	325	55
Equivalent mutants	4	31	22
Mutants killed by new counterexample	4	11	6
Mutants with Eldarica timeout	0	1	0
Invalid mutants	1	21	14
Number of test cases	4	6	4
Time for generating mutants	0:00:39	0:00:02	0:00:01
Time for analyzing mutants	0:01:20	0:12:30	0:04:44
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 22. Results for the program “Reverse a given number” with an error that the last digit is ignored if it is a 1

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	28	468	122
Mutants killed by previous test cases	20	385	60
Equivalent mutants	1	7	21
Mutants killed by new counterexample	3	27	7
Mutants with Eldarica timeout	2	5	1
Invalid mutants	2	44	33
Number of test cases	3	14	7
Time for generating mutants	0:00:37	0:00:03	0:00:02
Time for analyzing mutants	0:03:39	0:26:34	0:13:59
Time for generating final test cases	0:00:04	0:00:03	0:00:05
Error found by test cases?	yes	yes	yes

Table 23. Results for “C Program to reversing an Array Elements in C Programming” with an error that the middle two elements are ignored

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	27	496	186
Mutants killed by previous test cases	7	238	36
Equivalent mutants	0	13	23
Mutants killed by new counterexample	18	163	42
Mutants with Eldarica timeout	0	6	0
Invalid mutants	2	76	85
Number of test cases	6	9	8
Time for generating mutants	0:00:32	0:00:03	0:00:02
Time for analyzing mutants	0:02:51	0:38:30	0:15:05
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 24. Results for "C Program to read integers into an array and reversing them using pointers" with an error that the result array is filled with copies of only one element

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	39	493	136
Mutants killed by previous test cases	18	414	74
Equivalent mutants	5	23	26
Mutants killed by new counterexample	6	15	11
Mutants with Eldarica timeout	0	3	0
Invalid mutants	10	38	25
Number of test cases	4	11	9
Time for generating mutants	0:00:48	0:00:04	0:00:02
Time for analyzing mutants	0:02:42	0:16:21	0:08:21
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 25. Results for the program "Searching element in array" with an error that the position of any element not equal to the searched value is returned

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	16	258	52
Mutants killed by previous test cases	12	226	18
Equivalent mutants	0	1	11
Mutants killed by new counterexample	3	3	3
Mutants with Eldarica timeout	0	0	1
Invalid mutants	1	28	19
Number of test cases	3	3	3
Time for generating mutants	0:00:25	0:00:02	0:00:01
Time for analyzing mutants	0:00:27	0:03:57	0:03:14
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	no	no

Table 26. Results for the program "Find the simple interest" with a rounding error

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	31	651	124
Mutants killed by previous test cases	5	369	46
Equivalent mutants	2	24	20
Mutants killed by new counterexample	19	65	21
Mutants with Eldarica timeout	3	116	11
Invalid mutants	2	77	26
Number of test cases	10	22	12
Time for generating mutants	0:00:34	0:00:03	0:00:02
Time for analyzing mutants	0:07:03	2:30:19	0:23:41
Time for generating final test cases	0:00:02	0:00:03	0:00:02
Error found by test cases?	yes	yes	yes

Table 27. Results for "C program to find Smallest Element in Array in C Programming" with an error that the first element is not considered

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	3	340	84
Mutants killed by previous test cases	0	276	51
Equivalent mutants	0	14	13
Mutants killed by new counterexample	2	7	4
Mutants with Eldarica timeout	0	0	0
Invalid mutants	1	43	16
Number of test cases	2	6	4
Time for generating mutants	0:00:11	0:00:03	0:00:01
Time for analyzing mutants	0:00:18	0:08:35	0:03:35
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	no	yes	yes

Table 28. Results for "C Program to calculate Addition of All Elements in Array" with an error that the last element is ignored

	Dextool Mutate	MUSIC	Universal Mutator
Total number of mutants	32	539	128
Mutants killed by previous test cases	24	481	50
Equivalent mutants	0	1	19
Mutants killed by new counterexample	7	11	9
Mutants with Eldarica timeout	0	0	0
Invalid mutants	1	46	50
Number of test cases	7	11	9
Time for generating mutants	0:00:38	0:00:04	0:00:02
Time for analyzing mutants	0:00:54	0:08:17	0:05:15
Time for generating final test cases	0:00:02	0:00:02	0:00:02
Error found by test cases?	yes	yes	yes

Table 29. Results for the program "Calculate sum of 5 subjects and Find percentage" with a rounding error