Cornelius: Killing Mutants with E-graphs

Ben Kushigian
Yihong Zhang
Ishan Chatterjee
Gabrielle Strandquist

ABSTRACT

Mutation analysis is a powerful technique that seeds faults, called mutants, into a program to determine a test suite's sensitivity to changes in the subject under test. However it suffers from scalability problems. One of the barriers to mutation analysis' adoption in practice is the equivalent mutant problem. Equivalent mutants are seeded faults that differ syntactically from the original program but are semantically equivalent.

The presence of equivalent mutants introduces a source of error in common metrics used to measure test suite sensitivity. Furthermore, triaging and addressing non-obvious equivalent mutants wastes developer time adding to the burden of implementing mutation testing. Therefore, equivalent mutant detection has been an active area of research. One promising direction is the use of compiler techniques to rewrite equivalent mutants and the original program to syntactically-equivalent programs, making equivalence trivial to detect.

This approach suffers from two challenges. The first challenge is that rewrite rules can produce exponentially many syntactic variants of a program, and thus it is possible that two programs that can in theory be rewritten to the same form under a rewrite system end up in different forms. The second challenge is that the order in which rewrite rules are applied matters: two equivalent programs may be rewritten in an order such that their equivalence is never witnessed. This is a variant of the well-known *phase ordering problem*.

To address these issues, we propose to use the E-graph data structure to track all equivalent variations of a program and its mutants simultaneously. We implement Cornelius, a prototype of this idea, and evaluate it on simple Java expressions. We compare Cornelius' performance with Trivial Compiler Equivalence (TCE) using the Soot optimization framework, as well as with Medusa, which uses an SMT solver and symbolic execution to detect equivalent mutants. We find Cornelius discovers more equivalent and redundant mutants than TCE on our dataset, in less time than both TCE and Medusa.

KEYWORDS

E-graphs, equivalent mutant detection, equality saturation

ACM Reference Format:
Ben Kushigian, Yihong Zhang, Ishan Chatterjee, and Gabrielle Strandquist. 2020. Cornelius: Killing Mutants with E-graphs. In *Proceedings of CSE 503: Software Engineering*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CSE 503: Software Engineering, March 20, 2020, Seattle, WA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Suppose you wanted to prove that two programs were equivalent—how would you do it? One way would be to prove that they produce the same output for all possible inputs. Depending on the program this is possible, and may even be automated. In general, however, this can be expensive at best, and is often intractable.

Another method to prove program equivalence is to apply sequence semantics-preserving syntactic rewrites to the two programs and compare the resulting programs for syntactic equality. Since the rewrites are semantics-preserving, any rewritten programs that become syntactically identical under these rewrites must have begun as semantically equivalent programs. Optimizing compilers perform such rewrites to generate better[1] versions of the same program.

The problem of detecting equivalent programs arises in *mutation analysis*, a test suite analysis technique that seeds faults, or *mutants*, in a program to determine if a test suite can detect them. A mutant is detected if a test that passes on the original program fails on the mutant, and we say that the mutant is *killed* by the test.

Some mutants are semantically equivalent to the original program and cannot be killed by any test, even in theory—these mutants are called *equivalent mutants*. Other mutants may be semantically distinct from the original program, but semantically equivalent to another mutant—these are called *redundant mutants*. We want to detect equivalent and redundant mutants to improve mutation analysis's accuracy and efficiency.

Prior work [12] suggests that simply looking at the output of optimizing compilers might be sufficient to detect some equivalent mutants. This approach has two problems:

- First, rewrite rules often produce many syntactic variants of a program, and it is possible that two equivalent programs are optimized into different syntactic forms and their equivalence is never witnessed, even though the rewrite system is powerful enough to witness program equivalence.
- Second, an optimizing compiler must *choose* which rewrite rule to apply at a given point in time. This choice may preclude other rewrites from being triggered, causing some program variants to never be discovered; this is known as the *phase ordering problem*[15].

When comparing programs we would like to simultaneously compare *all* of their syntactic variants that are reachable via a rewrite system, and we would like to do so efficiently. One way to do this is to use an *E-graph*, a union-find data structure originally used by constraint solvers to solve problems in the theory of equality and uninterpreted functions[4].

[1]a better program may be more efficient, have less assembly code, have fewer method calls, have fewer memory accesses, etc

When a rewrite rule is applied, the E-graph stores not only the newest program, but also the original program, employing space-saving tactics to make this efficient. Using E-graphs to aid with program rewrites has been explored by Tate et al in the context of optimizing compilers[13]. We discuss E-graphs and rewrite systems further in Section 3.

In Section 4 we introduce Cornelius, a tool employing E-graphs and rewrite systems to detect equivalent and redundant mutants in Java expressions. Cornelius presently only works on total Java expressions over integers and booleans, and it is future work to generalize the technique.

In section 5 we measure Cornelius' ability to detect equivalent and redundant mutants against other tools for the same purpose. We also examine Cornelius' time performance in comparison to these tools.

We present the results of our evaluation in Section 6.

2 BACKGROUND ON MUTATION ANALYSIS

Mutation analysis has been demonstrated to be a valuable tool in determining the effectiveness of a given test suite to detect real world faults [5].

Mutation analysis is a fault-based technique where syntactic changes are made to a program. These syntactic changes result in new programs called *mutants*. Mutants are proxies for real world faults, and a test suite's ability to identify, or *kill*, the generated mutants is used as an approximation for its ability to identify real world bugs. Different forms of mutation testing *kill* differently. We focus on *strong mutation testing* where a test kills a mutant when the original program passes the test and the mutant fails the test.

Each mutant is a proxy for a bug, and each mutant killed by our test suite is a proxy for our test suite catching a bug. This is measured with the *mutation kill ratio* which is defined as the ratio between the number of mutants killed and the number of mutants generated. If all mutants are killed the mutation kill ratio is 1; if no mutants are killed, the mutation kill ratio is 0.

While prior work suggests that mutation detection correlates with real fault detection [5], computing the mutation kill ratio is expensive. One reason is that many equivalent and redundant mutants are generated. An *equivalent mutant* is a mutant that is semantically equivalent to original program. If an equivalent mutant is used in the analysis, the test suite will be run on the mutant in an attempt to kill it, even though this is impossible. This wastes CPU cycles. Further, a developer may try to write a test to kill the mutant, only to discover that it is equivalent. This wastes developer cycles. Finally, if equivalence is never discovered, it will count as a living mutant and count against the test suite's efficacy at detecting mutants, skewing analysis results.

Other mutants may be semantically distinct from the original program, but semantically equivalent to another mutant—these are called *redundant mutants*. Unlike equivalent mutants, redundant mutants can be killed, but killing them gives us no new information since they are equivalent to another mutant, and killing them wastes CPU cycles. What's more, killing a mutant kills all other mutants equivalent to it, and killing a set of redundant mutants can skew analysis results.

Thus we want to detect equivalent and redundant mutants to improve mutation analysis's accuracy and efficiency. While detecting equivalent or redundant mutants is undecidable [2], there are many cases of equivalence that can be caught statically.

3 REWRITE SYSTEMS AND E-GRAPHS

3.1 Rewrite Systems

A rewrite system consists of a language L and a set of rewrite rules R. A *rewrite rule* $r \in R$ comprises a *pattern* that can match terms in L and an *applier* that transforms terms matching the pattern to a new term in the language. Note that a rewrite doesn't need to be applied to an entire term, but can also be applied to a subterm. If a term s can be rewritten to a term t under rewrite system R we write $s \rightsquigarrow_R t$. When R is clear from context, we write $s \rightsquigarrow t$.

For instance, consider the *Sums* language

```
Sums   ::= Expr
Expr   ::= var
         |  int-lit
         |  (+ Expr Expr)
```

with the following rewrite rules

`(+ a b)` $\mapsto$ `(+ b a)`	commutativity
`(+ (+ a b) c)` $\mapsto$ `(+ a (+ b c))`	associativity
`(+ 0 a)` $\mapsto$ `a`	zero-identity

Consider terms $t_1 =$ `(+ (+ 0 a) (+ b c))` and $t_2 =$ `(+ c (+ b a))`. $t_1 \rightsquigarrow t_2$:

`(+ (+ 0 a) (+ b c))`		
$\rightsquigarrow$	`(+ a (+ b c))`	**(zero-identity)**
$\rightsquigarrow$	`(+ a (+ c b))`	**(commutativity)**
$\rightsquigarrow$	`(+ (+ c b) a)`	**(commutativity)**
$\rightsquigarrow$	`(+ c (+ b a))`	**(associativity)**

However, the $t_2 \not\rightsquigarrow t_1$ since there is no rewrite rule to introduce a zero.

3.1.1 Inducing equivalence relations. Viewing $\rightsquigarrow$ as a binary relation between terms in the language, we can construct an equivalence relation $\leftrightsquigarrow$ that respects $\rightsquigarrow$; this can be explicitly constructed by taking the transitive, symmetric, and reflexive closure of $\rightsquigarrow$. Thus, $t_1 \leftrightsquigarrow t_2$ even though $t_2 \not\rightsquigarrow t_1$.

Stated differently, $s \leftrightsquigarrow t$ precisely when there exists a sequence of terms $t_1, t_2, \ldots, t_n$, where $s = t_1$ and $t = t_n$, such that for each $1 \le i < n$, either $t_i \rightsquigarrow t_{i+1}$ or $t_{i+1} \rightsquigarrow t_i$.

3.1.2 Semantics-preserving rewrite systems. Many languages have some sort of associated *semantics*. For instance, a term in the *Sums* can be thought of as a program that takes bindings from variables to integer literals and produces a sum of integers. We are especially interested in *semantics-preserving* rewrite systems: rewrite systems that preserve the semantics of the language that they are rewriting.

Under our chosen semantics for *Sums*, the above rewrite system is semantics-preserving since each rewrite rule is a well-known arithmetic identity.

For the remainder of this paper we assume that all rewrite systems are semantics-preserving.

3.2 The Program Equivalence Relation

Programs P_1 and P_2 are semantically equivalent, written $P_1 \equiv P_2$, if every input produces the same output:

$$\forall x : P_1(x) = P_2(x).$$

For a rewrite system R, each equivalence class of the $\leftrightsquigarrow$ relation is a subset of an equivalence class of the $\equiv$ relation; we say that $\leftrightsquigarrow$ *respects* $\equiv$. We have the following theorem

Theorem 3.1. *Let R be a semantics-preserving rewrite system over language L. Then $\leftrightsquigarrow_R$ respects $\equiv$.*

Proof. Let P_1, P_2, be programs of L, and suppose that $P_1 \leftrightsquigarrow_R P_2$. By definition of $\leftrightsquigarrow_R$ there is a sequence of programs $Q_1 = P_1, \ldots, Q_N = P_2$ such that for each $1 \le i < N$ either $Q_i \rightsquigarrow_R Q_{i+1}$ or $Q_{i+1} \rightsquigarrow_R Q_i$. Since $\rightsquigarrow_R$ is semantics-preserving, each $Q_i \equiv Q_{i+1}$, and by transitivity all $Q_i \equiv Q_j$. Thus, any two elements of an equivalence class of the $\leftrightsquigarrow_R$ relation must be in the same equivalence class of the $\equiv$ relation, and $\leftrightsquigarrow_R$ respects $\equiv$. □

3.3 E-graphs

At its core, detecting equivalent and redundant mutants with rewrite systems boils down to efficiently tracking equivalence classes of programs, a task very much suited to an E-graph. An *E-graph* is a union-find data structure that efficiently computes and stores a congruence relation of an expression tree.

Recall that in union-find, each equivalence class has a *canonical representative*, a member of that class through which all comparisons are made. Thus, if I want to determine if elements x and y are equivalent, I ask the union-find structure if $x.canonical = y.canonical$.

We motivate with an example. Consider the programs $p_1 = (+\ 1\ a)$ and $p_2 = (+\ 1\ b)$, and suppose that we find out from a rewrite that $a \leftrightsquigarrow b$. We would like to conclude that $p_1 = (+\ 1\ a) \leftrightsquigarrow (+\ 1\ b) = p_2$. A union-find structure would represent this as $p_1 = (+\ 1\ \langle a, b\rangle)$ and $p_2 = (+\ 1\ \langle a, b\rangle)$, where $\langle a, b\rangle$ is the equivalence class containing a and b.

While p_1 and p_2 appear the same to us, each + symbol began with different children and thus were initially treated as separate nodes by union-find. When a and b are merged through a rewrite, union-find never checks their parents to see if they can be merged, and thus p_1 and p_2 remain distinct: $p_1.canonical \ne p_2.canonical$.

It turns out that the traditional union-find data structure is not quite up to the task, and we need to modify it to check for possible merges of nodes' parents. E-graphs add this functionality.

An E-graph consists of *E-classes*, which are equivalence classes of E-nodes. An *E-node* is similar to a node in an AST: it has a *label* such as +, −, *foo*, etc., but rather than having other E-nodes as children, it has *E-classes* as children. Additionally, different parents can *share* E-classes as children, and an E-node or an E-class may have multiple parents. For instance, the program `(+ (- 1 a) (+ 1 a))` will only have a single E-node for the value `a`, and a single E-node for the value `1`. This practice, called *node sharing*, reduces space and time costs.

```
int max(int a, int b) {          int max(int a, int b) {
  return a > b ? a : b;            return a < b ? a : b;
}                                }


int max(int a, int b) {          int max(int a, int b) {
  return a >= b ? a : b;           return a == b ? a : b;
}                                }
```

Figure 1: The `max` function and three of its mutants

Each E-class has a canonical representative, just like in union-find, but when two E-classes are merged, the parents of one class are checked to see if they can be merged with any parents of the other class.

E-nodes may be thought of as functions whose children are their arguments (E-nodes with no children are constant functions). With this view, E-graphs extend traditional union-find algorithms with the *congruence axiom*, which states that if f is a function and if for all $1 \le i \le n$, $a_i = b_i$, then $(f\ a_1\ a_2\ \ldots\ a_n) = (f\ b_1\ b_2\ \ldots\ b_n)$.

By detecting when two instances of a given function symbol are applied to the equivalent arguments, an E-graph can conclude that the instances of the function symbol themselves are equivalent. Take, for instance, the programs `(+ a (* 2 2))` and `(+ a 4)`. The operator +, and values `a` and `2` are each repeated twice. An E-graph will store both of these programs simultaneously and efficiently by sharing equal nodes: each instance of a will be shared, as will each instance of the node 2. By applying rewrite rule `(* a b)` $\rightsquigarrow a \cdot b$, nodes 4 and `(* 2 2)` are discovered to be equivalent. The E-graph then applies the congruence axiom, inferring that the two instances of the + operator must in turn be equivalent.

When an E-graph reaches a point where no rewrite rules an be applied to create new equivalences we say that the E-graph has reached *equality saturation.* Upon reaching equality saturation, an E-graph represents every possible program that can be obtained the original E-graph by the rewrite system. *egg* runs until either equality saturation is reached, or until predetermined resources are exhausted. When equality saturation is reached, or when *egg* has used as much time or space as allotted to it, we can query the E-graph to determine if two programs are in the same equivalence class.

We propose to use E-graphs and an appropriate rewrite system to detect equivalent and redundant mutants. By storing the original program and its mutants in an E-graph and computing equality saturation of the rewrite system, we expect to identify a large portion of equivalent and redundant mutants.

4 IMPLEMENTATION

The E-graph is central to our implementation but is difficult to implement correctly. We chose to use the *egg* library [16]. *egg*, and thus Cornelius, is written in Rust.

Cornelius expects as input an `xml` file containing a list of serialized *subjects*. Each subject consists of an original program and a list of mutants of that program, all in a Lisp-like form, called an *egg-expression*, that can easily be fed to *egg*. For instance, the program `max` in Figure 1 and its three mutants would be encoded as

```
<subject method="max(int,int)">
    <egg>(? (> a b) a b)</egg>
    <mutant id="1">
        <egg>(? (< a b) a b)</egg>
    </mutant>
    <mutant id="2">
        <egg>(? (>= a b) a b)</egg>
    </mutant>
    <mutant id="3">
        <egg>(? (== a b) a b)</egg>
    </mutant>
</subject>
```

For each subject Cornelius creates an E-graph, parses the egg-expressions from the subjects original program and its mutants, and stores egg-expressions in the E-graph. Cornelius does some bookkeeping, tracking ids assigned to each program by *egg* so that it can ask about equivalence relations after running the rewrite rules.

Currently, Cornelius supports pure and total Java expressions on primitive `int` and `boolean` types. We refer to this fragment of Java as *expr-lang*. In particular, division and remainder operators, method calls, and impure operators such as post-increments, are not allowed.

While this is limiting, it is a good first step towards handling arbitrary Java code. For instance, to handle the following program:

```
int max(int a, int b) {
    int max = a;
    if (b > max) max = b;
    return max;
}
```

Cornelius would first translate into a static single assignment (SSA) form:

```
int max(int a, int b) {
    int max0 = a;
    max1 = $\phi(b > max_0, b, max_0)$
    return max1;
}
```

This can be translated into a single expression:

$\phi(b > a, b, a)$

4.1 Cornelius' Rewrite Rules

Cornelius has a large number of rewrite rules, broken into the following categories:

- **Arithmetic Rewrites** commutativity and associativity of + and *, subtraction, basic arithmetic identities, etc
  - `(+ x y)` ⇝ `(+ y x)`
  - `(* x y)` ⇝ `(* y x)`
  - `(+ x (+ y z))` ⇝ `(+ (+ x y) z)`
  - `(* x (* y z))` ⇝ `(* (* x y) z)`
  - `(+ 0 x)` ⇝ `x`
  - `(* 0 x)` ⇝ `0`
  - `(+ (- x) x)` ⇝ `0`
  - `(- x y)` ⇝ `(+ x (- y))`

  For now we don't include division since we need reason about handling exceptional behavior for dividing by zero, but this won't be particularly difficult to implement.
- **Logical Rewrites** commutativity and associativity of `&&` and `||`, as well as simple logical rules such as
  - `(|| x y)` ⇝ `(|| y x)`
  - `(&& x y)` ⇝ `(&& y x)`
  - `(|| x (|| y z))` ⇝ `(|| (|| x y) z)`
  - `(&& x (&& y z))` ⇝ `(&& (&& x y) z)`
  - `(&& true x)` ⇝ `x`
  - `(&& false x)` ⇝ `false)`
  - `(|| true x)` ⇝ `x`
  - `(&& false x)` ⇝ `false)`
- **Ordering Rewrites** this section includes rules for rewriting ordering operators, such as
  - `(< a b)` ⇝ `(! (>= a b))`
  - `(>= a b)` ⇝ `(|| (> a b) (== a b))`
- **Branching Rewrites** this category handles rewrites of conditional expressions. For instance,
  - `(? true t e)` ⇝ `t`
  - `(? false t e)` ⇝ `e`
  - `(? c t e)` ⇝ `(? (! c) e t)`
- **Constant Propagation** this category handles constant propagation rewrites
- **Equality Refinement** this category handles terms of the form `(? (== a b) thn els)`:
  - `(? (== a b) t e)` ⇝ `(? (== a b) (swap a b t) e`
  - `(? (== a b) t e)` ⇝ `(? (== a b) (swap b a t) e`
- **Swap Rewrites** This category of rewrite rules recursively applies swaps to arguments of operators as well as swapping variables where appropriate
  - `(swap a b a)` ⇝ `b`
  - `(swap a b (> l r))` ⇝ `(> (swap a b l) (swap a b r))`
  - `(swap a b (+ l r))` ⇝ `(+ (swap a b l) (swap a b r))`
  - `(swap a b (&& l r))` ⇝ `(&& (swap a b l) (swap a b r))`

5 METHODOLOGIES

With Cornelius built, we pose the following research questions:

RQ1 How many equivalent and redundant mutants does Cornelius detect compared to other equivalent and redundant mutant detection tools?

RQ2 How fast does Cornelius run compared to other automated equivalent and redundant mutant detection tools?

To evaluate these questions we collected subjects, mutated them, and ran Cornelius on each subject and its mutants. We also ran Medusa and Trivial Compiler Equivalence on all subjects as baselines. We describe these in further detail below. This section details the design and construction of our experiment, and we have made all materials open source[2].

5.1 Subject collection

We evaluated Cornelius on several subjects drawn from the *expr-lang* Java fragment described in Section 4. Each subject is a single expression wrapped in the return statement of a method; there is precisely one subject per method. The `max` method from Figure 1 is one such subject. Subjects are divided among three files:

[2]https://gitlab.cs.washington.edu/benku/cornelius

(1) `Expr.java` contains 14 subjects. These tend to be smaller expressions, some of which were crafted to have equivalent or redundant mutants, and some of which were created to explore different linguistic features of *expr-lang*.
(2) `TriangleExpr.java` adapts the classic `Triangle` program to *expr-lang*. This subject has a single method that classifies a triangle as equilateral, isosceles, scalene, or invalid, based on the lengths of its sides.
(3) `Misc.java` consists of two methods, `foo` and `bar`. `foo` is a single giant disjunction and was written with the purpose of generating a lot of equivalent mutants. `bar` is similar to `foo` but has some disjunctions flipped to conjunctions and is in conjunctive normal form.

5.2 Mutant generation

We used the Major mutation framework [6] to mutate the subjects described in 5.1.

To generate mutants we ran Major with all mutations that didn't produce possibly-partial expressions. In particular, we didn't produce division or remainder operators since these can possibly produce exceptional behavior, and *expr-lang* consists of total programs. This is automated and can be reproduced by running the following from the root of the Cornelius repository:

```
cd subjects
JAVA_HOME=/path/to/java-8 ./mutate.sh
```

This produced 115 mutants for `Exprs.java`, 47 mutants for `TriangleExpr.java`, and 202 mutants for `Misc.java`.

5.3 Subject Serialization

Next we serialized each subject and its mutants. Major produces a `mutants.log` that summarizes each mutant generated, including a unique numerical identifier (i.e., 1, 2, 3, ...) for each mutant, and provides some meta-data such as the method that the mutant belonged to.

We parsed `mutants.log` to identify which mutants corresponded to which methods in each file. We used JavaParser [9] to parse each original method and its corresponding mutants, as identified in `mutants.log`, and serialized these to XML files in Cornelius' input format, as specified in Section 4. This can be automated by executing the following command from the root of the Cornelius repository:

```
cd serialization
./serialize.sh
```

5.4 Running Cornelius

We compiled Cornelius with optimizations and ran it using the `time` utility. Cornelius's processing time consists of the AST parsing with JavaParser, generating E-graphs for the subjects and the process of running rewrite rules on the E-graphs to detect equivalence classes. The times do not include the processing times for Major to generate mutants. From the root of the Cornelius repository, we ran:

```
cd cornelius
./run.sh
```

This builds an optimized version of Cornelius and then uses the `time` utility to time an execution on each of the three programs described in Section 5. Timing info is printed to `stderr` and the files `data/exprs.txt`, `data/misc.txt`, and `data/triangle-expr.txt` contain the per-subject equivalence relation discovered by Cornelius.

5.5 Running Medusa

Medusa is a mutant detection framework built on top of Z3. Medusa constrains Java bytecode by implementing portions of the JVM with SMT constraints. To check for program equivalence, Medusa constrains both programs, asserts that the inputs are equal, and asserts that the outputs are not equal. Then it asks Z3 if this is satisfiable. If this is satisfiable, then there must exist inputs that produce different outputs for the two programs, and they are not equivalent. Otherwise, if this is unsatisfiable, the programs are equivalent since all inputs yield to the same outputs for both programs.

We chose to use this as a benchmark for two reasons. First, it uses a fundamentally different approach to that of TCE or Cornelius. Second, it can be used for ground truth since it is both sound and complete on *expr-lang*; that is, it will always produce the correct output. This soundness and completeness comes from the fact that total Java expressions on booleans and 32-bit signed integers can be translated into the theory of fixed-width bit vectors, and Z3 implements a decision procedure for this theory.

It is possible that Medusa times out since Medusa is solving NP-hard problems, but if such a timeout occurs than more time can be allocated and the query can be rerun. We have not witnessed a timeout so far.

Medusa can be downloaded, installed, and run on the subjects by invoking `./medusa.sh` from the root of the Cornelius repository. This will print times to `stderr` and discovered equivalences and redundancies to `stdout`. This takes several hours to run.

5.6 Running Trivial Compiler Equivalence

We chose Trivial Compiler Equivalence (TCE) as our second baseline since this is the approach most similar to Cornelius'. The Javac compiler is not optimizing, so after compilation we run optimizations from the Soot[14] framework on the produced bytecode.

This is automated and can be done by running `tce.sh` from the root of the Cornelius repository. Since both Java compilation and Soot are internal to TCE, both of these processes are included in the collected times for TCE reported in Section 6. In future work we might refactor TCE to allow for just the running of Soot to obtain more accurate times.

5.7 Equivalent mutants, redundant mutants, and equivalent programs

Fundamentally, each of the three equivalent and redundant mutation detection techniques is approximating the program equivalence relation $\equiv$ presented in Section 3; in Medusa's case, this is a perfect approximation for *expr-lang*.

Importantly, there is a subtlety with how we collect and present our data. Consider the following case. Suppose that mutants m_1 and m_2 are equivalent mutants; that is, that they are equivalent to each other and that each is equivalent to the original program p. Then there are three equivalences that can be witnessed: $p \equiv m_1$, $p \equiv m_2$, and $m_1 \equiv m_2$. Suppose that only the third equivalence was witnessed by Cornelius. Ground truth says that m_1 and m_2 are equivalent mutants, but Cornelius reports that m_1 and m_2 are *redundant* mutants. From this interpretation, not only has Cornelius missed an equivalent mutant, but it has also falsely reported a redundant mutant. By this measure it appears to have performed worse than if it had never discovered that $m_1 \equiv m_2$. TCE also suffers from this same issue.

To address this, we instead report *equivalent mutants* and *equivalent programs*. Equivalent mutants are precisely as we have been defining them, while equivalent programs comprise *all* reported equivalences of a program. In particular, equivalent mutants are a subset of equivalent programs, as are redundant mutants.

For each tool, the reported number of *equivalent mutants EM* for a subject is the size of the equivalence class of the original program reported by that tool minus 1:

$$EM(Tool, orig) := |\text{equiv-class}(orig)| - 1.$$

In our above example, Medusa would report that p's equivalence class was $\langle p, m_1, m_2 \rangle$ which has size three; thus $EM(Medusa, p) = 2$. Cornelius would report that p's equivalence class is $\langle p \rangle$, and thus $EM(Cornelius, p) = 0$.

Likewise, the reported *equivalent programs* for a subject is the sum of $|C|$ minus 1 over all equivalence classes C reported by a tool:

$$EP(Tool) := \sum_{C \in \textit{equiv-classes(Tool)}} (|C| - 1)$$

6 RESULTS

6.1 Detection of Equivalent Mutants and Equivalent Programs

As a baseline, we ran Medusa and TCE on each subject, and we compared the *EM* and *EP* between each tool.

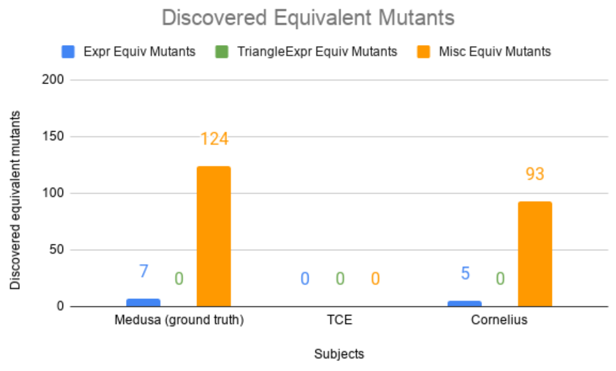

Figure 2: Detected equivalent mutants by each tool. Detected equivalent mutants are split across java programs to show the distribution of reported equivalent mutants per each program.

Cornelius detects the majority of equivalent mutants and equivalent programs. For equivalent mutants across all subjects, Medusa found 131 out of the total 364 mutants to be equivalent to the original program, while Cornelius detected 98, yielding a recall of 74.8% (see Table 3). For equivalent programs across all subjects, Medusa detected 172 while Cornelius found 123, yielding a recall of 71.5%.

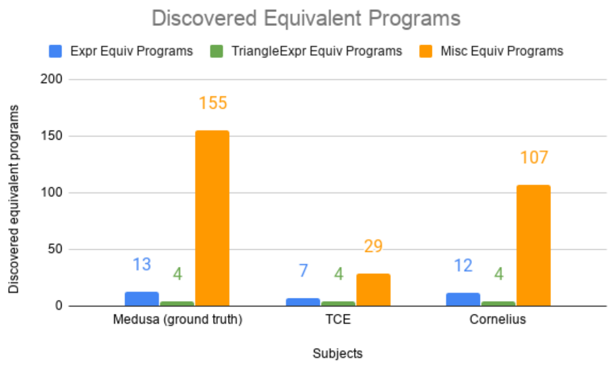

Figure 3: Detected equivalent programs by each tool. Detected equivalent programs are split across java programs to show the distribution of reported equivalent programs per each program.

We briefly examine two equivalent mutants in one of `Expr.java`'s subjects that Cornelius failed to detect. The original method, `multByTwo`, is as follows:

```
public int multByTwo(int a) {
    return a == 2 ? 2 + a : 2 * a;
}
```

Cornelius failed to catch mutants 18 and 19 from Expr.java:

```
public int multByTwo(int a) {
    return false ? 2 + a : 2 * a;
} //mutantID 18

public int multByTwo(int a) { 9
    return a == 2 ? 2 * a : 2 * a;
} //mutantID 19
```

These equivalent mutants were missed because of the way Cornelius implements equality refinement. When a conditional expression is reached, if the condition is an equality check, say `a == 2`, Cornelius uses this information to help reason about the *thn* and *els* clause. To do this, Cornelius propagates a `swap` rewrite rule that switches all instances of `a` to `2` and all instances of `2` to `a`. Note that these are done *simultaneously*:

```
(? (= a b) thn els) =>
    (? (= a b) (swap a b thn) els)
```

These are propagated down the AST with rewrite rules such as

```
(swap a b (+ x y)) =>
    (+ (swap a b x) (swap a b y))
(swap a b a) => b
(swap a b b) => a
```

This means that `(+ a 2)` can be rewritten to `(+ 2 a)`, but not to `(+ 2 2)`, where our constant folding rewrite rules can be applied. One possible fix is to add a rewrite rule

```
(swap a b c) => c
```

Table 1: Detected Equivalent Mutants

Java Program	Total Mutants	Total Equivalent Mutants	Discovered Equivalent Mutants
Exprs	115	7	5
TriangleExpr	47	0	0
Misc	202	124	93

Table 2: Detected Equivalent Programs

Java Program	Total Mutants	Total Equivalent Programs	Discovered Equivalent Programs
Exprs	115	13	12
TriangleExpr	47	4	4
Misc	202	155	107

Table 3: Cornelius Precision and Recall

	Recall	Precision
Equivalent Mutants	74.8%	100%
Equivalent Programs	71.5%	100%

However, this fails because this allows the form (swap a b a) to rewrite to both a and b. But this is a *global* rewrite, and the E-graph unifies all instances of a and b which is wrong.

An alternative solution, which is only partial, is to rewrite to constants whenever possible. This would work in the above case but cannot generalize to cases that contains complicated equality information between variables. Instead, we need to enrich our E-graph expressions to include the path constraints satisfied to reach a given location. How this can be done is an open problem, but there has been some interest stirring in the E-graph community[3].

As was expected, Cornelius reported no false positives (100% precision) for detecting both equivalent mutants and equivalent programs.

6.2 Processing times

Table 4 compares the processing time of Cornelius to the processing times of TCE and Medusa.

As can be seen in Table 4, the processing times for Medusa and TCE are on the order of minutes while Cornelius's times are within a few seconds, showing a clear advantage for Cornelius. Medusa's long run times are the result of two factors. First, solving SMT queries is hard and expensive. Second, Medusa has to do pairwise comparisons of programs, asking for each pair of programs if they are equivalent. Cornelius does not have to do this since all programs are stored in an E-graph, and node sharing allows for a single rewrite rule to apply to many programs simultaneously.

While TCE is faster than Medusa, it still has to compare $O(n^2)$ Java bytecode outputs after compiling the programs. Additionally, it has to invoke the full Javac compiler and Soot framework on each program, and while these step are not very expensive on their own, they add up.

[3]i.e., Max Willsey is interested.

6.3 Discussion

While initial results are very promising, we stress that we are working with a restricted set of the Java language. Additionally, the implementation of TCE that we executed using Javac and Soot mainly uses dataflow propagation which may not be the most effective method to catch equivalent mutants and equivalent programs in *expr-lang*. An avenue of future work is to translate these expressions to C and use TCE with the gcc optimizing compiler.

Even though these results don't generalize to arbitrary mutants, there still may be use cases to quickly detect equivalent mutants of expressions before they are generated, and another avenue of future work is to study how often mutated expressions are equivalent to the original expression.

7 RELATED WORKS

Many researchers have tried to solve the equivalent mutant problem. Budd and Angluin showed that detecting equivalent mutants is undecidable [2]. Researchers have tried to apply constraint solvers to detect equivalent mutants [3, 8, 10], but these approaches have not scaled to production code.

Baldwin and Sayward [1] first proposed using compiler optimizations to detect equivalent mutants in 1979. Offutt and Craft used dataflow analysis to rewrite mutants with the hope that equivalent mutants will be rewritten to the same program [11]. They reported finding 45% of the equivalent mutants generated, though 100 of the 114 detected equivalent mutants were generated by the absolute value insertion operator that is not used by Major; their technique detected 16% of the equivalent mutants generated by other operators. The authors only worked on small programs and did not consider combinations of techniques, looking only at a single technique at a time. They also did not try to detect redundant mutants.

In 2015 Papadakis et al [12] introduced trivial compiler equivalence (TCE) by using the gcc optimizing compiler to detect equivalent and redundant mutants in real-world C programs. This technique caught a large number of equivalent mutants generated by the ABS, UOI, ROR, and CRCR mutation operators. While the CRCR and ROR operators seem to produce a large proportion of useful mutants, the ABS and UOI operators seem to produce a large number

Table 4: Processing Times (seconds)

Java Program	Medusa real-time	TCE real-time	Cornelius real-time
Exprs	280	302	1.4
TriangleExpr	370	112	0.6
Misc	8080	488	0.9

of equivalent mutants. In total they discarded over 7% of generated mutants as equivalent and over 21% of mutants as redundant.

Kintes et al [7] followed this work by applying dataflow analysis to Java mutants. Since Javac is not an optimizing compiler the authors used the Soot framework [14] to perform dataflow operations on Java bytecode. The authors reported discarding 5.7% of generated mutants as equivalent and 5.4% of mutants as redundant. Again, most of the identified mutants were generated by problematic mutation operators that are not used by Major.

While TCE seems like it may be promising, there is no guarantee that two mutants will be rewritten to the same version of the program, despite there existing some ordering of rewrites that could. This is known as the *phase ordering problem* [15], and is a well known problem in optimizing compilers. To address the phase ordering problem, Tate et al proposed *equality saturation* [13], where different program versions are tracked simultaneously in an E-graph. An E-graph is a data structure that efficiently computes and stores congruence closures of an equivalence relation. Tate et al produced the tool PEGGY which works on Java control flow graphs.

8 CONCLUSIONS

This paper introduces Cornelius, a framework that employs E-graphs and rewrite rules to detect equivalent and redundant mutants and presents a prototype that can soundly reason about mutant equivalence for *expr-lang*, a restricted fragment of the Java programming language.

By utilizing E-graphs, a data structure that can efficiently compute and store rewrites of a program, our approach was able to discover a greater number of equivalent mutants than TCE on subjects in *expr-lang* by avoiding the *phase ordering problem*. Furthermore, we show that our approach performs much faster on the same set of subjects than both TCE via Soot and Medusa.

While limited in the set of expressions it can currently evaluate, Cornelius demonstrates a promising direction for further investigation. Rewrite rules related to control flow such as branching and loops are present in optimizing compilers now, and incorporating these rules would allow Cornelius to expand the set of program mutants in can efficiently analyze.

With scalable and efficient equivalent mutant detection, we hope to reduce the burden of deploying mutation testing, paving the way to more widespread use.

REFERENCES

[1] Douglas Baldwin and Frederick Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations.* Technical Report. GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE.

[2] Timothy A Budd and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta informatica* 18, 1 (1982), 31–45.

[3] R.A. DeMilli and A.J. Offutt. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17, 9 (Sept. 1991), 900–910. https://doi.org/10.1109/32.92910

[4] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification*, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Rajeev Alur, and Doron A. Peled (Eds.). Vol. 3114. Springer Berlin Heidelberg, Berlin, Heidelberg, 175–188. https://doi.org/10.1007/978-3-540-27813-9_14 Series Title: Lecture Notes in Computer Science.

[5] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, Hong Kong, China, 654–665. https://doi.org/10.1145/2635868.2635929

[6] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 612–615.

[7] Marinos Kintis, Mike Papadakis, Yue Jia, Nicos Malevris, Yves Le Traon, and Mark Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* 44, 4 (April 2018), 308–333. https://doi.org/10.1109/TSE.2017.2684805

[8] Benjamin Kushigian, Amit Rawat, and Rene Just. 2019. Medusa: Mutant Equivalence Detection Using Satisfiability Analysis. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Xi'an, China, 77–82. https://doi.org/10.1109/ICSTW.2019.00035

[9] Matazoid. 2007. Javaparser. https://github.com/javaparser/javaparser

[10] Simona Nica and Franz Wotawa. 2012. Using Constraints for Equivalent Mutant Detection. *Electronic Proceedings in Theoretical Computer Science* 86 (July 2012), 1–8. https://doi.org/10.4204/EPTCS.86.1 arXiv: 1207.2234.

[11] A. Jefferson Offutt and W. Michael Craft. 1994. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 4, 3 (1994), 131–154. https://doi.org/10.1002/stvr.4370040303

[12] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 936–946. https://doi.org/10.1109/ICSE.2015.103 ISSN: 1558-1225.

[13] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. [n.d.]. EQUALITY SATURATION: A NEW APPROACH TO OPTIMIZATION. ([n. d.]), 80.

[14] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[15] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *ACM SIGMICRO Newsletter* 13, 4 (Dec. 1982), 125–133. https://doi.org/10.1145/1014194.800942

[16] Max Willsey. 2020. egg: E-graphs Good. https://github.com/mwillsey/egg