Medusa: Mutant Equivalence Detection Using Satisfiability Analysis

Benjamin Kushigian
University of Massachusetts
Amherst, MA, USA
bkushigian@cs.umass.edu

Amit Rawat
University of Massachusetts
Amherst, MA, USA
amitrawat@cs.umass.edu

René Just
University of Washington
Seattle, WA, USA
rjust@cs.washington.edu

***Abstract*—This paper introduces Medusa, a framework for reasoning about the equivalence of first-order mutants in Java programs. Since the problem of detecting equivalent mutants is undecidable in general, even when restricted to first-order mutants, Medusa focuses on a subset of Java that can be modeled as SMT solver constraints. This paper describes the key insights behind Medusa and provides details about its concepts and components, in particular constraint forking—a novel approach that leverages structural similarities between mutants to improve its efficiency. This paper further reports on a preliminary evaluation and outlines several optimizations that leverage the first-order mutant property to further improve Medusa's applicability and efficiency.**

***Index Terms*—Mutation testing, equivalent mutants, SMT solver**

I. Introduction

Mutation analysis is widely used in research [1]–[3], and mutation-based testing sees increasing adoption in practice (e.g., [4], [5]). However, equivalent mutants remain a major concern as they skew analysis results and waste resources.

Recent advances in SMT solvers [6] make automatically generating proofs of mutant equivalence and model checking non-equivalence a viable option [7]–[9]. By constructing a first-order logical formula φ representing the execution of a program p and a mutant m, we may ask an SMT solver if there is a set of inputs for which p and m yield different outputs. If not, φ is *unsatisfiable*, and m is an equivalent mutant and can be discarded. Otherwise, m is a non-equivalent mutant, and we may ask the SMT solver for inputs that witness non-equivalence, thereby generating a test that kills m. In both cases the mutation score increases: in the first case the increase corresponds to an improved accuracy; in the second case the increase corresponds to an improved test suite.

SMT-based mutant equivalence detection and test generation suffer from two major challenges: *applicability* (whether or not an approach can reason about a program) and *efficiency* (the amount of resources needed to reason about a program).

Applicability challenges stem from programming language features, such as heap allocation, loops, method calls, and exceptions, which occur in real-world programs [10], [11]. Many of these features cannot be defined by first-order logic—they are not *FO-definable*—and thus cannot be modelled with SMT solvers. In practice, however, many special cases of undecidable problems can be reasoned about[1]. Additionally, we can improve applicability by abstracting the execution of non-FO-definable features into FO-definable ones. Medusa is motivated by the fact that while many problems are undecidable in general, instances that are encountered in the wild tend to follow certain patterns that can be reasoned about. For example, Java generics are Turing complete [12], but developers rarely, if ever, leverage this fact. Hence, applicability can be improved by identifying and solving frequently encountered special cases of these problems.

Efficiency challenges stem from the fact that, at their core, SMT solvers are solving instances of *SAT*, a well known NP-complete problem. This paper hypothesizes and demonstrates that additional information about inputs to the equivalent mutant problem, such as the first-order mutation property, can be used to increase efficiency by refining the queries made to the SMT solver.

This paper introduces Medusa, a framework for automatically reasoning about mutant equivalence by modelling JVM bytecode with SMT constraints. The current prototype of Medusa can prove mutant equivalence in loop-, heap-, and call-free programs using primitive types. Medusa also handles some object references and restricted cases of method invocations such as boxing and unboxing primitives.

To improve Medusa's efficiency, we designed, implemented, and evaluated *constraint forking*, a new approach that leverages structural similarities between mutants to efficiently reason about batches of mutants (Sections V and VI).

To improve Medusa's applicability, we propose a new approach called *middle-out constraint generation* (Section VII) as well as some additional techniques (Section VIII). Middle-out constraint generation allows Medusa to reason about subsets of a method execution, such as loop bodies, without constraining the entirety of the method.

II. Background

Mutation analysis measures a test suite's ability to distinguish a program under test from many artificial faults—small variations, called *mutants*. A test that detects a mutant is said to *kill* that mutant. Given a program p, mutation analysis applies a set of small syntactic changes (*mutations*) to p, generating a set of mutants M. A *first-order mutant* is one derived from applying precisely one mutation; we say that such a mutant satisfies the *first-order mutation property*.

[1]For example, the halting problem is known to be undecidable in general, but the program `int foo(){return 17;}` can be easily proven to halt.

The necessary conditions for a test t to kill a mutant $m \in M$ can be described using the reachability, infection, and propagation (RIP) model [11]:

1) Reachability: t must execute m's mutation at least once.
2) Infection: at least one execution of m's mutation must cause m's execution state to differ from that of p.
3) Propagation: the infected execution state of m must propagate to some observable output.

There are two common variants of mutation analysis: weak and strong mutation. In *weak mutation*, a test t kills a mutant m if t satisfies the infection condition. In *strong mutation*, a test t kills a mutant m if t satisfies the propagation condition. Note that satisfying the reachability condition is equivalent to satisfy statement coverage of the mutated statement. This paper is concerned with strong mutation.

A mutant that cannot be killed by any test is called an *equivalent mutant*. The *mutation score* $\mu(T)$ of a test suite T is defined as the number of mutants killed by T divided by the number of generated nonequivalent mutants. To compute $\mu(T)$, all equivalent mutants must be identified, an intractable problem. Instead, $\mu(T)$ is commonly approximated by the *mutation kill ratio* of T, defined as the number of killed mutants divided by the number of generated mutants that have not (yet) been identified as equivalent.

Each unidentified equivalent mutant wastes resources and underestimates $\mu(T)$. The problem of identifying equivalent mutants is thus crucial to improving the accuracy and efficiency of mutation analysis.

III. Generating Constraints from Bytecode

This section describes our approach to modelling program execution with SMT constraints, enabling queries to SMT solvers about mutant equivalence. As a running example, suppose we want to reason about the equivalence of mutants `m1` and `m2` to the original program `squarePos` in Figure 1a. We are first confronted with the problem of defining the program semantics. Since the semantics are ultimately defined by Javac and the JVM, we work with JVM bytecode instead of Java source code.

A. SMT Constraints

A *constraint* is a boolean expression in a representation understood by our constraint solver. In the remainder of this paper, constraints are written in prefix syntax. For example, `(ite (< x y) (= z x) (= z y))` represents a constraint for `z = x < y ? x : y`. We can *constrain* a program p by generating constraints that capture p's execution; we will also speak of constraining blocks and edges of p's control flow graph.

We may *assert* a set of constraints to our constraint solver—that is, ask the solver to *check satisfiability*. The solver will attempt to find a model satisfying all asserted constraints and, if found, returns that model.

B. Control Flow Graph

The bytecode in Figure 1b is easier to reason about than the original source code, but the control flow is implicit; to make it explicit, Medusa calculates the program's control flow graph.

```
int squarePos(int x) {
  if (x > 0)
    x = x * x;
  return x;
}
```

Mutants	Equivalent
`m1: > ↦ >=`	yes
`m2: > ↦ <`	no

(a) Original source code and two mutants m1 and m2.

```
B0: 0: iload_1
    1: ifle 8   (B2)
B1: 4: iload_1
    5: iload_1
    6: imul
    7: istore_1
B2: 8: iload_1
    9: ireturn
```

(b) Bytecode (m1 and m2 differ at instruction 1).

```
Setup Parameters in Table:
  (= vt1-0 x)
B0:
  iload_1: push to stack
  (= (tail s1) s0)
  (= (head s1) vt1-0)
  ifle: set cond, pop stack
  (= B0-cond (<= (head s1) 0))
  (= s2 (tail s1))
B1:
  iload_1: load x to stack
  (= (tail s4) s3)
  (= (head s4) vt1-1)
  iload_1: load x to stack
  (= (tail s5) s4)
  (= (head s5) vt1-1)
  imul: multiply, push
  (= (tail s6) s3)
  (= (head s6) (mul (head s4)
                    (head s5)))
  istore_1: store to vt
  (= vt1-2 (head s6))
  implicit branch condition
  (= B1-cond true)
B2:
  iload_1: load x
  (= (tail s8) s7)
  (= (head s8) vt1-3)
```

(c) Block constraints

```
Was Block 0 executed?
(=> B0-executed?
  (ite B0-cond
    True Branch:
    (and
      Propagate to B2:
      (= B2-executed? true)
      (= vt1-3 vt1-0)
      (= s7 s2))
    False Branch:
    (and
      Propagate to B1:
      (= B1-executed? true)
      (= vt1-1 vt1-0)
      (= s3 s2))))

Was Block 1 executed?
(=> B1-executed?
  (ite B1-cond
    (and
      Propagate to B2:
      (= B2-executed? true)
      (= vt1-2 vt1-3)
      (= s7 s6))))

Was Block 2 executed?
(=> B2-executed?
  Set return value
  (= ret (head s7)))
```

(d) Branching constraints

Fig. 1: Constraining a method from bytecode

A *control flow graph* (CFG) is a graph whose *edges* correspond to branching information and whose nodes, called basic blocks or just *blocks*, are maximal straightline sequences of bytecode satisfying the following properties: (1) only the final instruction may be a jump instruction, and (2) only the first instruction of a block may be targeted by a jump instruction.

A block B may have either one or two successors. If there are two successors then there is an associated boolean condition c_B such that the *true successor* is executed after B when c_B and the *false successor* is executed after B when $\neg c_B$. If B has only one successor then Medusa sets c_B to be `true` and considers B's only successor to be its true successor.

Medusa constrains blocks and edges separately, which allows for the flexibility needed in constraint forking and middle-out constraint generation.

C. Block Constraints

Recall our running example in Figure 1a. Medusa has generated the CFG, comprising blocks B0, B1, and B2. Figure 1b shows the labelled bytecode, which reflects this partitioning. Medusa then uses each of these blocks to generate constraints of the call-stack state, shown in Figure 1c. The call-stack has two types of state: the operand stack (*opstack*) and the variable table (*vartable*). Note that Medusa does not track heap state.

To represent the JVM opstack at a point of execution as a constraint, Medusa uses linked lists, which support operations `head` and `tail`; vartable entries are represented as width-32 bit-vectors. When pushing value `x` onto the opstack, represented by linked list `s`, Medusa creates a new linked list instance `s′` and generates the following constraints:

```
        (and (= (head s′) x) (= (tail s′) s))
```

Each time Medusa pushes to or pops from the opstack it introduces a static single assignment (SSA) variable to represent the newest version of the opstack. Likewise, whenever Medusa stores a value to the vartable it introduces a new SSA variable. Medusa names variables as follows: `s2` represents the third SSA linked list variable generated to track the opstack (names are zero-indexed), while `vt1-3` represents the fourth SSA variable generated for the first vartable entry. Figure 1c follows these conventions and is annotated for convenience.

Finally, each block has an execution bit to constrain control flow. Medusa uses the notation `executed-2?` to denote the value of the predicate "was Block 2 executed?".

D. Edge Constraints

Medusa connects the generated block constraints by asserting *edge constraints* based on the last instruction of each block. If the last instruction is a conditional jump, as in B0, then the head of the final opstack is used to compute a branching condition. Otherwise the branching condition is set to `true` to indicate that the block's true successor is always taken. Medusa then uses this condition to determine control flow and propagate a block's final state to its successor's initial state. Edge constraints take the following form:

```
(=> B-executed? (ite B-cond
     (and (propagate B Bt) executed-Bt?)
     (and (propagate B Bf) executed-Bf?)))
```

`Bt` and `Bf` are `B`'s true and false successors. In other words, these constraints translate to: *If `B` was executed and `B`'s branching condition is true, propagate `B`'s state to its true successor and set that successor's execution bit to `true`. Otherwise, if `B` was executed and `B`'s branching condition is false, propagate `B`'s state to its false successor and set that successor's execution bit to `true`.* Figure 1d shows the edge constraints generated by Medusa for our example.

E. Checking Mutant Equivalence

To test for equivalence between a program p and a mutant m, Medusa constrains the execution of both p and m, asserts that their inputs are equal, and asserts that their outputs are different. Continuing with our example, let `m1-x` be `m1`'s input and `m1-ret` be `m1`'s return value. Then, to test if `squarePos` and `m1` are equivalent, Medusa asserts:

```
        (and (= x m1-x) (not (= ret m1-ret))).
```

If the solver returns *SAT*, it has found a model witnessing non-equivalence; if it instead returns *UNSAT*, it has proven equivalence. The solver may also return *UNKNOWN*, e.g., if it times out. For `m1`, the solver returns *UNSAT* while for `m2`, it returns *SAT*, providing a model witnessing a difference (e.g., `x:=-1`). The provided model can be used for test generation.

IV. Medusa Strategies

Medusa works on batches of mutants, and to take advantage of this we introduce the notion of a *strategy*, allowing Medusa to tailor its approach to the batch of mutants it is reasoning about. This in turn allows Medusa to reuse some of its work and apply different approaches to different types of mutants.

The *naive strategy* is the strategy that checks for equivalence between program `p` and each of its mutants `m` using the approach described in Section III-E. The naive strategy does not reuse any of its work between mutants, recomputing `p`'s constraints and all lemmas derived from `p`'s constraints for each of `p`'s mutants.

The central SMT feature that enables work reuse is the *scope*. Scopes may be *pushed* and *popped* in a LIFO fashion, and an assertion is made into the most recently pushed scope. The solver cannot produce a model violating this assertion until that assertion's scope is popped. Popping a scope clears all assertions made in that scope as well as all lemmas generated since it was pushed.

We get immediate performance gains by refactoring the naive strategy to generate constraints for p only once, pushing and popping a new scope to reason about each mutant. We call this the *caching strategy*, and while it yields substantial improvements, as detailed in Section VI, we can do better.

V. Constraint Forking

Medusa has not leveraged the fact that it is working with first-order mutants. To help refine queries to the SMT solver, we define an *atomic mutant* to be a mutant with at most one syntactic element changed. This definition implicitly depends upon the program representation (i.e., source code, AST, bytecode, CFG, etc.). Many first-order mutants result in *atomic CFG mutants*, mutants in which at most one block may be altered, no blocks are deleted, and all edges and their true/false labels fixed. For instance, removing a short circuiting operator removes a jump instruction and results in a non-atomic CFG mutant, whereas negating a conditional alters the control flow but not the CFG structure, and thus results in an atomic CFG mutant.

Execution is identical between a program and a mutant until a mutated block is reached. Hence, the beginning of both executions can be modeled as the same constraints. To capitalize on this, we propose the *forking* strategy.

First, Medusa generates $\mathcal{C}_p$, the constraints for p, as well as an identical copy $\mathcal{C}_m$ of $\mathcal{C}_p$, which Medusa will use to constrain mutants. Since the execution of p is always the same, Medusa asserts $\mathcal{C}_p$, but since the execution of each mutant is different, Medusa does not immediately assert $\mathcal{C}_m$.

Second, Medusa labels all blocks $B_1, \ldots, B_N$ such that blocks precede their parents, and creates an empty stack S of blocks. Then, Medusa iterates through all blocks $B_1, B_2, \ldots, B_N$:

1) Push an SMT scope.
2) Push B_i onto S.
3) Assert B_i's block and outgoing edge constraints in $\mathcal{C}_m$.

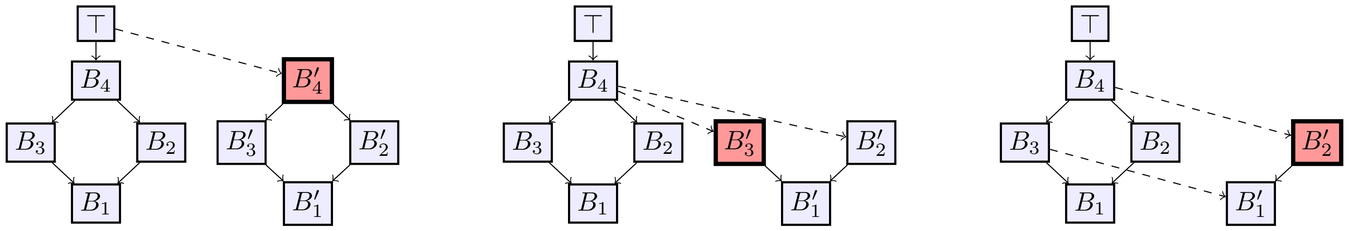

Fig. 2: Progression of the forking strategy. Mutated blocks are highlighted (red), and dashed lines represent forked constraints. For each atomic CFG mutant of a block B_i, constraints are forked from B_i's predecessor to the mutated block B'_i.

Medusa asserts that $\mathcal{C}_p$ and $\mathcal{C}_m$'s return values are different. Once this initial pre-pass is completed Medusa compares mutant m_i with p. While S is not empty:

1) Let $B = S.pop()$.
2) Pop a scope, clearing B's edge constraints in $\mathcal{C}_m$.
3) For each mutated block B' of B, do:
    a) Push a scope.
    b) Generate edge constraints from B' into $\mathcal{C}_m$.
    c) Generate edge constraints from $\mathcal{C}_p$ to B'.
    d) Call `(check-sat)` and, if *SAT*, call `(get-model)`.
    e) Pop the scope, clearing B''s constraints.

Figure 2 illustrates three iterations of this process. Each mutated block, highlighted in red, corresponds to multiple atomic CFG mutants.

VI. Evaluation

We evaluated Medusa on a Linux workstation with Intel Xeon CPU at 2.2GHz and 100GB of RAM. Each execution of Medusa was limited to one CPU and at most 2GB of RAM.

We evaluated Medusa on three subjects:

1) **Tax:** Computes the single-payer tax amount for a given income.
2) **TicTacToe:** Checks the win condition for the Tic Tac Toe game, including bounds checking on inputs.
3) **Triangle:** Classifies a given triangle into equilateral, isosceles, scalene, or invalid.

We chose the above programs for their branching structure since techniques such as loop unrolling and method inlining, which Medusa could apply in a preprocessing step, tend to result in a large amount of branching. (Real-world methods often use language features such as references, method calls, and loops that Medusa cannot yet reason about. Section VIII addresses this applicability concern.)

We used MAJOR [13] to generate a total of 488 mutants across all three subjects: 122 for Triangle, 267 for TicTacToe, and 99 for Tax. Table I provides a breakdown of the three subjects, including size in bytecode instructions, the total number of mutants, and the total number of equivalent mutants.

We manually determined ground truth for mutant equivalence. The naive and the caching strategies correctly calculated equivalence for all mutants, and the forking strategy correctly calculated equivalence for all atomic CFG mutants.

TABLE I: Summary of the three evaluation subjects. **Insns** and **Jmps** give the total number of bytecode instructions and the number of jump instructions. For **All mutants** and **Atomic CFG mutants**, **tot** and **equiv** give the total number of mutants and the number of equivalent mutants.

Subject	Insns	Jmps	All mutants		Atomic CFG mutants	
			tot	equiv	tot	equiv
Tax	100	15	99	10	84	10
TicTacToe	171	49	267	23	98	19
Triangle	89	17	122	4	70	3

TABLE II: Run times (in seconds) and improvements for the **naive**, caching (**cache**), and forking (**fork**) strategies. Run times are averaged over five runs.

Subject	Run times			Improvements	
	naive	cache	fork	cache (vs. naive)	fork (vs. naive)
Atomic CFG mutants					
Tax	521	258	197	50.5%	62.2%
TicTacToe	17.9	16.5	13.9	7.8%	22.5%
Triangle	3.72	3.26	1.3	12.4%	65.1%
Non-atomic CFG mutants					
Tax	410	185	—	54.9%	—
TicTacToe	17	16.3	—	4.1%	—
Triangle	2.34	2.16	—	7.7%	—
*All mutants**					
Tax	504	247	195	51.0%	61.3%
TicTacToe	17.4	16.3	15.4	6.3%	11.5%
Triangle	3.13	2.79	1.67	10.9%	46.6%

*For non-atomic CFG mutants, the forking strategy defaults to caching.

We computed the average run times of the naive, caching, and forking strategies on all mutants per subject over five executions. Table II summarizes the results and reports on the improvements in run time when using caching or forking in place of the naive strategy. Improvements are calculated to be $100 \cdot (t_n - t_s)/t_n$, where t_n is the run time for the naive strategy, and t_s is the run time for the strategy being compared. In contrast to the other subjects, Tax primarily uses floating point arithmetic, which is likely the reason for its longer run times.

```
int gcd(int a, int b){       int gcd_mut(int a, int b){
  while (a != b)               while (a != b)
    if (a > b) a = a - b;        if (a >= b) a = a - b;
    else b = b - a;              else b = b - a;
  return a; }                  return a; }
```

Fig. 3: Middle-out generation can help with loops

When restricted to atomic CFG mutants, forking improves run time between 22.5% and 65.1%, and caching improves run time between 7.8% and 50.5%. The improvements across all mutants reflect this same pattern.

Both the naive and caching strategies took longer to determine (non-)equivalence for atomic CFG mutants than they did for other mutants. We conjecture that Medusa can more easily reason about non-atomic CFG mutants, which often involve changes in control flow structure such as the deletion of a short-circuiting boolean operator. We leave a deeper investigation open for future work.

VII. Middle-Out Constraint Generation

While mutant equivalence in the presence of loops is undecidable, there are mutants whose equivalence is clear. Consider `gcd` and its mutant `gcd_mut` in Figure 3. These are identical in all but the relational operator in the `if` statement's condition and execution is altered only if `a == b`. In this case, the loop's body, whose execution is predicated on `a != b`, could not have run, and we conclude that `gcd_mut` is equivalent.

Because of the presence of the `while` loop we cannot fully generate constraints for `gcd` or `gcd_mut`. This motivates *middle-out constraint generation*, where only part of a method is constrained. The key idea is as follows: check if a block and a mutated block are equivalent and if so conclude that both methods are equivalent. Otherwise expand checking to the immediate predecessors and successors and repeat. Continue until equivalence is proven or until expansion cannot continue.

In our example we begin by constraining the mutated blocks comprising the conditional `a > b` and mutation `a >= b`, querying if there exists a tuple `(a, b)` that distinguishes the two. Our solver returns *SAT* and we continue by constraining the predecessors or the successors. For brevity we expand up, generating constraints for `a != b`. We query if there exist a tuple `(a, b)` such that `a != b` and `a == b` which is impossible, hence our solver returns *UNSAT*.

VIII. Additional Techniques

In addition to constraint-forking and middle-out constraint generation, we propose several other techniques to increase Medusa's applicability.

Generalized Constraint Forking While the forking strategy applies to atomic CFG mutants, we conjecture that most first order mutants exhibit strong locality properties, even after translation to the control flow graph, and that this fact may be used to generalize the forking strategy.

Dynamic Loop Unrolling Loop unrolling is typically implemented as a static analysis prepass, which is orthogonal to our proposed methods. For example, Nica and Wotawa increase nesting depth for the entire program if non-equivalence was not proved and regenerate constraints from scratch [10]. We conjecture that dynamically unrolling loops by adding new iterations to a loop as needed while querying the SMT solver may increase applicability and efficiency. Dynamic loop unrolling would generate constraints only for new iterations, and only as needed; the remainder of the constraints would remain intact, and any work done by the SMT solver up to that point will be preserved. We expect dynamic loop unrolling to pair nicely with middle-out constraint generation.

Exception Abstraction Given Java's propensity for `NullPointerException`s, Java code is often riddled with null checks, and manually writing tests for these is tiresome. Consider the three null checks in the `nullCheck` method below.

```
int nullCheck(Object a, Object b, Object c) {
  if (a==null || b==null || c==null)
    throw new Exception();
  ... }
```

Adequately testing this method requires one test per null check. Even though this method contains heap allocation, method invocation, and exception throwing, Medusacould handle this and similar instances with an abstraction: in a prepass it could recognize when an exception is created and thrown and, instead of constraining the exact bytecode, emit constraints for an `exception-thrown` bit. We call this process of abstracting over exception handling *exception abstraction*. The SMT solver can prove non-equivalence if it finds a model in which one program throws and propagates an exception but the other does not. However, due to the abstraction, Medusa does not prove equivalence if both programs throw an exception—one program may throw an `IOException`, while the other may throw a `NullPointerException`.

Trivial Mutant Detection Just as equivalent mutants are a burden for developers [14], non-equivalent mutants also present a challenge. While many non-equivalent mutants are easy to kill, creating tests that do so can be tedious, time-consuming, and even unproductive [5]. One particular example are *trivial mutants*: mutants that throw an exception every time they are executed [15]. For instance, mutating a bounds check from `i<a.length` to `i<=a.length` in a loop that accesses the `i`th element of array `a` will always throw an exception; such a mutant provides no value beyond simple code coverage measures and should not be factored into the mutation score. Employing SMT solvers along with exception abstraction to aid in the detection of trivial mutants is promising future work.

Foldability In functional programming there is the notion of a *foldable* function which takes a list of data `xs`, an accumulator `acc`, and some combination function `f`, and recursively applies `f` to both the head of `xs` and `acc`, "folding" the results into the accumulator. An example of a foldable function is `sum`, which may be written as `sum xs = fold + 0 xs`, where `+` is the function that adds two integers. Given a foldable program `p xs = fold f acc xs` and a mutant `m` of `p` of the form `m xs = fold f' acc xs`, it is easy to see that if `f` is equivalent to `f'` then `m` is equivalent to `p`.

Foldable functions are common, though Java's syntax obfuscates this, and by recognizing them we will be able to reason about language features that are not FO-definable. Consider the `clip` method below, which maps non-negative numbers to themselves and negative numbers to zero:

```
int[] clip(int[] a) {
  int[] b = new int[a.length];
  for (int i=0; i < a.length; ++i)
    b[i] = a[i] < 0 ? 0 : a[i];
  return b;
}
```

Each cell of the output array `b` depends upon precisely one cell of `a` and on no other non-constant values. We may write `b[i] = f(a[i])`, where `f` is the function `f(x) = x < 0 ? 0 : x`. If we mutate this to `f′(x) = x <= 0 ? 0 : x` then this is an equivalent mutant: the only input which yields a local state infection is `0`, but `f(0) = f′(0)`. We conclude that both `clip` and its mutant are equivalent, even though they operate on arrays, which are not FO-definable.

IX. Related Work

The problem of equivalent and redundant mutants has been attacked from several angles [1], including static approaches [7], [8], [10], [11], [16], dynamic approaches [17]–[19], and hybrid approaches [20]. This section discusses the static approaches most closely related to ours.

DeMillo and Offutt [11] first identified *reachability*, *infection*, and *propagation* (RIP) as conditions for a mutant to be non-equivalent. If any of these conditions fail, the mutant is equivalent. Offutt and Pan [16] use constraint solvers to statically reason about mutant equivalence, considering the RIP model. They use heuristics to identify infeasible constraints in a mutation system to determine mutant equivalence. While these approaches are sound, they were hampered by the limited forms of constraints they could reason about. Access to powerful SMT solvers allows Medusa to reason about more complex constraints and improve over this seminal work.

Nica and Wotawa introduced EqMutDetect [10], which uses the MINION constraint solver and improves over [16]. EqMutDetect unrolls a program's loops, converts the program to SSA form, and translates this form into constraints for the solver. Medusa's naive strategy is roughly equivalent to this approach, modulo loop unrolling. This approach is sound for loop free programs; soundness for looping programs is only guaranteed for non-equivalent mutants. Medusa improves over EqMutDetect with novel optimization strategies (caching and forking) and additional features such as handling some object references and restricted cases of method invocations (e.g., boxing and unboxing primitives).

Compiler techniques and optimizations have been applied to the equivalent mutant problem in [21], [22], and more recently in [23]. The latter applies trivial compiler equivalence (TCE) to Java. TCE is predominately effective at identifying equivalent mutants generated by the AOIS operator, which introduces pre- and post-increment/decrement operators to variables. Due to the high number of equivalent mutants generated by this operator, it is not implemented in MAJOR.

X. Conclusion

This paper introduces Medusa, a framework that employs constraint solving techniques to detect equivalent mutants, and presents a prototype that can soundly reason about mutant equivalence for a restricted set of methods. It also proposes and details constraint forking and middle-out-constraint generation, two new techniques to increase efficiency and applicability of SMT-based equivalent mutant detection. This paper further describes additional promising techniques, whose implementation and evaluation is left open for future work.

Equivalent mutant detection is a hard problem but we are convinced that significant progress is possible with the use of modern SMT solvers, by taking advantage of structural knowledge gleaned from the first-order mutation property and by solving frequently occurring special cases.

References

[1] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of ICSE*, 2005, pp. 402–411.

[3] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. of FSE*, 2014, pp. 654–665.

[4] H. Coles, "Real world mutation testing," Online, http://pitest.org, last accessed January 2019.

[5] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, and R. Just, "An industrial application of mutation testing: Lessons, challenges, and research directions," in *Proc. of Mutation*, 2018, pp. 47–53.

[6] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*. Springer, 2008, pp. 337–340.

[7] B. K. Aichernig, E. Jöbstl, and M. Kegele, "Incremental refinement checking for test case generation," in *Tests and Proofs*, 2013, pp. 1–19.

[8] E. Jöbstl, "Model-based mutation testing with constraint and smt solvers," Ph.D. dissertation, 2014.

[9] N. Tillmann and P. de Halleux, "Pex - white box test generation for .net," in *TAP*, vol. 4966. Springer Verlag, April 2008, pp. 134–153.

[10] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," *ArXiv e-prints*, Jul. 2012.

[11] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE TSE*, vol. 17, no. 9, 1991.

[12] R. Grigore, "Java generics are turing complete," in *Proc of ACM SIGPLAN*, vol. 52, no. 1. ACM, 2017, pp. 73–85.

[13] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. of ASE*, 2011, pp. 612–615.

[14] B. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proc. of Mutation*, 2009, pp. 192–199.

[15] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proc. of ISSTA*, 2017, pp. 284–294.

[16] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *JSTVR*, vol. 7, no. 3, pp. 165–192.

[17] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proc. of ISSTA*, 2014, pp. 315–326.

[18] R. Gopinath, C. Jensen, and A. Groce, "Topsy-turvy: a smarter and faster parallelization of mutation analysis," in *Proc. of ICSE*, 2016, pp. 740–743.

[19] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proc. of ISSTA*, 2017, pp. 295–306.

[20] R. Just, M. D. Ernst, and G. Fraser, "Using state infection conditions to detect equivalent mutants and speed up mutation analysis," in *Proc. of Dagstuhl Sem.*, vol. abs/1303.2784, 2013, arXiv:1303.2784.

[21] D. Baldwin and F. Sayward, *Heuristics for Determining Equivalence of Program Mutations*, ser. Dept of CS: Research report, 1979.

[22] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *JSTVR*, vol. 4, pp. 131–154, 1994.

[23] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE TSE*, vol. 44, no. 4, pp. 308–333, 2018.