

What a Mutation Score Certifies

Adequacy-Completeness for Output Mutation

Author Name

Affiliation, City, Country
author@example.org

Abstract. Mutation testing scores a test suite by the fraction of seeded faults it distinguishes, and a score of 1 is read as evidence of adequacy. *Adequate for what* has never been answered: the field’s formal results characterise operator sets relative to a fixed program text or a fixed test suite, and its bridge from seeded faults to real ones rests on the coupling-effect hypothesis. We give a completeness notion that is neither program- nor suite-relative, for the class of *output mutation* operators (perturb the returned value; extreme or Descartes-style mutation is the constant special case). For adequacy such an operator is exactly its *footprint*, the set of values it changes. We call a family $\mathcal{I}$ adequacy-complete for a target family $\mathcal{F}$ when, for every program and every suite, killing all non-equivalent $\mathcal{I}$ -mutants forces killing all non-equivalent $\mathcal{F}$ -mutants, and prove an exact characterisation: completeness holds iff every target footprint is the union of the $\mathcal{I}$ -footprints it contains. Consequences: the minimal absolutely complete family on n values consists of the n singleton *value guards*, and none is finite on an infinite return type; a full absolute score certifies exactly output coverage and nothing more; first-order completeness does not entail higher-order completeness; completeness is decidable exactly when footprint containment is; and the minimum certifying suite has size $|f(D)|$. The development relativises to a weak oracle, yielding a precise characterisation of pseudo-tested methods that an empirical study of five Python libraries bears out: 43% of covered methods are pseudo-tested under a value oracle against only 6% under a crash-inclusive one. The core is mechanised in Lean 4 / Mathlib.

Keywords: Mutation testing · Test adequacy · Operator completeness · Extreme mutation · Interactive theorem proving

1 Introduction

Mutation testing [3,6] seeds a program with small faults —*mutants*—and measures a test suite by the fraction it *kills*, i.e. distinguishes from the original. A mutation score of 1 is the field’s adequacy certificate, and it is used as one: tools report it, teams gate on it, and a growing body of work uses it to judge machine-generated code.

The certificate’s strength rests entirely on the operator family. A score of 1 says that no fault *the operators can express* survived, and says nothing about

faults they cannot express. Two questions follow, and neither has a clean answer in the literature.

What does a score of 1 certify? Not correctness. The bridge from “no seeded fault survives” to “no real fault survives” is the *coupling-effect hypothesis* [3] and the *competent-programmer hypothesis*, both stated as motivating assumptions and supported empirically [12,7,8] rather than proven.

When is an operator family complete? The strongest available results are relative by construction. *Sufficient* operator sets [13] are determined experimentally against a chosen operator pool. *Minimal* mutant sets and *dominator* mutants [1] are minimal relative to a fixed test set and an already-generated pool. *True* (semantic) mutant subsumption—the order against which a completeness theorem would rank—is undecidable [9], and the foundational treatment of mutation adequacy [2] likewise establishes undecidability of the associated decision problems rather than a characterisation.

The relativity is not an oversight. A program-text mutation *schema* carries no operator-only invariant that fixes its detection uniformly across programs: the mutant it produces depends on the *source text* of the program, not on the function the program computes, so “does schema X subsume schema Y ” has no program-independent answer (Section 8). Any completeness notion for program-text operators must therefore quantify over programs *and* their texts, which is what forces the prior results to be relative-by-construction.

1.1 Approach

We restrict attention to a sub-class that does admit an operator-only invariant: **output mutation**, operators that perturb the value a function returns, independent of how the function computes it. Extreme mutation—replacing a method body with `return c`, as implemented in Descartes [11,15]—is the constant special case.

For this class we define completeness by *what a kill certifies*, quantified over all programs and all suites:

A family Π is **adequacy-complete for a target family Γ** iff, for every program and every finite suite, a mutation score of 1 against Π forces a mutation score of 1 against Γ .

Both relativities of the prior results are quantified away by the definition. The resulting theory is governed by a single combinatorial object—the *footprint* of an operator, the set of values it changes—and reduces to set containment.

1.2 Contributions

1. **The footprint reduction** (Section 3): for adequacy purposes an output operator *is* its footprint; equivalence and killing depend only on which values the operator moves, never on where it sends them.

2. **A characterisation theorem** (Section 4): Π is adequacy-complete for Γ iff every target footprint is the union of the Π -footprints contained in it. We give the theorem for finite Π (mechanised) and in a general form covering infinite families.
3. **The value-guard basis and the finite/infinite dichotomy** (Section 5): on n values the minimal absolutely complete family has size exactly n , consisting of the singleton *value guards*; on an infinite return type no finite output-operator family is absolutely complete. Descartes-style constants are complete iff $n = 2$.
4. **The ceiling** (Section 5): a full absolute score holds exactly when the suite exercises every reachable output. This is the whole operational content of output mutation, and it delimits the technique as much as it licenses it.
5. **Relative completeness and its decidability** (Section 6): completeness reduces to footprint containment, giving a P / decidable / undecidable spectrum that runs through set containment rather than through algebraic word problems.
6. **Higher-order mutants** (Section 7): first-order completeness does not entail higher-order completeness; the only positive statement is degenerate.
7. **Program independence** (Section 8): output-mutant detection factors through the footprint, which is exactly why a program-independent completeness notion is available here and not for program-text schemas.
8. **The minimum certifying suite** (Section 9): the cheapest certifying suite has size $|f(D)|$, the teaching dimension of the program against the operator family.
9. **Relativisation to a weak oracle** (Section 10): the entire development goes through with the footprint replaced by an oracle-relative detection set, yielding a precise characterisation of pseudo-tested methods.

The results of Sections 3–9 and the reduction from programs and finite suites down to footprints are mechanised in Lean 4 / Mathlib. Section 13 states exactly which results are machine-checked and which are paper proofs.

1.3 What this paper does not claim

The account is bounded, and the bounds are load-bearing rather than decorative. It concerns output mutation of pure total functions; it does not extend to general program-text mutation, and we make no claim about the coupling hypothesis for program-text faults. Sections 3.4 and 14 state the model’s assumptions and their consequences in full. The central quantitative result—that a full absolute score is equivalent to output coverage—is deliberately modest, and it implies that a tool gains no information from running an absolutely complete output-operator family rather than measuring output coverage directly. The contribution is a characterisation of what such a score means, not a recommendation to compute it.

2 Background and Related Work

Mutation testing. The technique originates with [3] and [6]; [7] and [14] survey its development. Its justification rests on two hypotheses: the *competent-programmer hypothesis*, that real faults are close to correct programs, and the *coupling effect*, that suites detecting simple faults tend to detect complex ones. The coupling effect has substantial empirical support [12,8] and no general proof.

Formal treatments. Budd and Angluin [2] give the foundational formal analysis of mutation adequacy, relating notions of correctness to testing and establishing undecidability results for the associated decision problems. Their results are undecidability theorems about a general setting, not a characterisation of which operator families make a passing score adequate; the present paper is complementary, giving such a characterisation for a restricted class where it is available.

Operator selection. Offutt et al. [13] identify *sufficient* operator sets experimentally, by measuring which subsets of a fixed pool retain the pool’s discriminating power on a corpus. Ammann et al. [1] formalise *minimal* mutant sets and *dominator* mutants: given a fixed program and a fixed test pool, the dominator mutants are those whose kill implies the kill of others. Kurtz et al. [9] study mutant subsumption and show that the true (semantic) subsumption relation is undecidable, restricting practice to computable approximations. All three are relative to a fixed program, a fixed pool, or both—which, as Section 8 argues, is forced by the absence of an operator-only invariant for program-text schemas.

Extreme mutation. Niedermayr et al. [11] introduce extreme mutation and the notion of a *pseudo-tested* method: one that is covered by the suite yet whose body can be replaced wholesale without any test failing. Vera-Pérez et al. [15] study the phenomenon at scale and provide the Descartes tool. In the terms of this paper, extreme mutation is the constant output-operator family; Section 5 recovers it as exactly optimal on two-valued return types and provably deficient beyond, and Section 10 gives a characterisation of pseudo-testedness in the model’s own terms.

Teaching dimension. The minimum number of labelled examples that identifies a concept within a class is the *teaching dimension* [5]. Section 9 identifies the minimum certifying suite with this quantity for the output-mutation setting.

Mechanisation. The development is checked in Lean 4 [4] against Mathlib [10].

3 The Model

3.1 Programs, Suites, Operators

Definition 3.1 (program, suite, output operator). *A program computes a total function—its denotation— $f : D \rightarrow R$, from an input domain D to a return*

type R with $|R| \geq 2$. A test suite is a finite $T \subseteq D$. An output operator is a total map $p : R \rightarrow R$; it produces the mutant $p \circ f$ (run the program, then apply p to the result). An output-operator family Π is a set of output operators.

Post-composition is what an output or extreme mutation performs: it perturbs the returned value uniformly, without reference to how the value was computed.

Definition 3.2 (footprint). *The footprint of an output operator p is*

$$\text{Mov}(p) = \{r \in R : p(r) \neq r\} \subseteq R,$$

*the set of values p actually changes. Write $I = f(D)$ for the program's **reachable** outputs and $O = f(T)$ for the suite's **observed** outputs; note $O \subseteq I$.*

Definition 3.3 (equivalence, killing, score). *For a program f and suite T :*

- $p \circ f$ is **equivalent** to f iff $p(f(x)) = f(x)$ for all $x \in D$;
- T **kills** $p \circ f$ iff $p(f(x)) \neq f(x)$ for some $x \in T$;
- the **score** $\text{score}_\Pi(f, T) = 1$ iff T kills every non-equivalent mutant $p \circ f$, $p \in \Pi$.

Equivalent mutants are excluded from the score, as in standard practice: they cannot be killed by any suite, so counting them would make a score of 1 unachievable.

3.2 The Footprint Reduction

Proposition 3.4 (an operator is its footprint). *For any program f , suite T , and operator p :*

$$p \circ f \text{ is equivalent} \iff \text{Mov}(p) \cap I = \emptyset, \quad T \text{ kills } p \circ f \iff \text{Mov}(p) \cap O \neq \emptyset.$$

*Consequently, for fixed (f, T) , whether p is a non-equivalent survivor depends **only** on $\text{Mov}(p)$; two operators with equal footprints are interchangeable; and $\text{score}_\Pi(f, T)$ depends only on the set of footprints $\text{Foot}(\Pi) = \{\text{Mov}(p) : p \in \Pi\}$.*

Proof. $p \circ f$ is equivalent iff p fixes $f(x)$ for every $x \in D$, i.e. p fixes every element of $I = f(D)$, i.e. no element of I is moved. T kills $p \circ f$ iff some $x \in T$ has $p(f(x)) \neq f(x)$, i.e. $f(x) \in \text{Mov}(p)$ for some $x \in T$, i.e. $\text{Mov}(p)$ meets $f(T) = O$. Both conditions read only $\text{Mov}(p)$ against the fixed sets I and O . $\square$

Proposition 3.4 is the engine of the paper. Where an operator *sends* the values it moves is invisible to adequacy; only *which* values it moves matters.

Proposition 3.5 (every subset is a footprint). *For $|R| \geq 2$, every $S \subseteq R$ is the footprint of some output operator.*

Proof. Fix distinct $a \neq b$. Let $q(a) = b$ and $q(x) = a$ for $x \neq a$; then q is fixed-point-free. Define $p(x) = q(x)$ for $x \in S$ and $p(x) = x$ otherwise. Then $\text{Mov}(p) = S$. $\square$

The construction avoids cycles: a single $|R|$ -cycle would not serve for uncountable R , whose orbits under iteration are countable. So footprints range over the entire powerset 2^R , and by Proposition 3.4 the adequacy content of a family Π is exactly $\text{Foot}(\Pi)$.

3.3 Notation

Throughout, Π denotes the operator family under test and Γ the target family whose faults we wish to certify against; $\mathcal{T}(R)$ denotes the family of *all* output operators on R , and $n = |R|$ when R is finite.

3.4 Scope and Assumptions

The model makes three assumptions. One is load-bearing and two are simplifying; we state them here, where the model is defined, and return to their consequences in Section 14.

Load-bearing: the exact oracle. Definition 3.3 kills a mutant exactly when some test yields a different return value, which presumes the suite observes the full output. Under a weaker oracle a covered mutant can survive because no assertion inspects the changed value—the *pseudo-tested method* phenomenon [11,15] that motivates extreme mutation in practice. Section 10 relativises the entire development to an arbitrary oracle and recovers the exact-oracle case as the special case where the oracle is injective; readers who care primarily about weak oracles may read Section 10 as the general theory and Sections 4–9 as its cleanest instance.

Simplifying: pure total functions. A denotation is a total map $f : D \rightarrow R$. Void methods, exceptions, non-termination, and side effects are outside the model. Extending to partial or effectful denotations requires enlarging R with the relevant observations (an exception tag, an effect trace), after which the footprint machinery applies unchanged to the enlarged type; we do not develop this.

Simplifying: uniform post-composition. An output operator perturbs every returned value identically, $f \mapsto p \circ f$. A statement-level mutation of a single **return** site is not of this form.

4 Adequacy-Completeness and the Characterisation Theorem

Definition 4.1 (adequacy-completeness). *Let Π and Γ be output-operator families. Π is **adequacy-complete for Γ** iff for every program $f : D \rightarrow R$ and every finite suite $T \subseteq D$,*

$$\text{score}_\Pi(f, T) = 1 \implies \text{score}_\Gamma(f, T) = 1.$$

Π is **absolutely complete** iff it is complete for $\Gamma = \mathcal{T}(R)$.

By Proposition 3.4 this depends only on $\text{Foot}(\Pi)$ and $\text{Foot}(\Gamma)$. The quantifier over all (f, T) removes both the program-text relativity and the test-suite relativity of prior completeness notions: the definition asks what a kill certifies *as such*, not what it certifies on some corpus.

Theorem 4.2 (footprint characterisation, finite Π). *Let Π be finite. Then Π is adequacy-complete for Γ iff*

$$\forall g \in \Gamma, \forall b \in \text{Mov}(g), \exists p \in \Pi : b \in \text{Mov}(p) \subseteq \text{Mov}(g).$$

Equivalently: every target footprint is the union of the Π -footprints it contains, $\text{Mov}(g) = \bigcup \{\text{Mov}(p) : p \in \Pi, \text{Mov}(p) \subseteq \text{Mov}(g)\}$.

Proof. ($\Leftarrow$) Assume the condition. Fix (f, T) with $\text{score}_\Pi(f, T) = 1$ and a non-equivalent $g \in \Gamma$; we show T kills $g \circ f$. Non-equivalence gives (Proposition 3.4) $\text{Mov}(g) \cap I \neq \emptyset$; pick $b \in \text{Mov}(g) \cap I$. By hypothesis there is $p \in \Pi$ with $b \in \text{Mov}(p) \subseteq \text{Mov}(g)$. Then $b \in \text{Mov}(p) \cap I$, so $p \circ f$ is non-equivalent, so—the Π -score being 1—it is killed: $\text{Mov}(p) \cap O \neq \emptyset$. Since $\text{Mov}(p) \subseteq \text{Mov}(g)$, also $\text{Mov}(g) \cap O \neq \emptyset$, i.e. T kills $g \circ f$.

($\Rightarrow$) By contraposition. Assume the condition fails, witnessed by $g \in \Gamma$ and $b \in \text{Mov}(g)$ such that every $p \in \Pi$ with $b \in \text{Mov}(p)$ has an *escape point* $a_p \in \text{Mov}(p) \setminus \text{Mov}(g)$. Let $A = \{a_p : p \in \Pi, b \in \text{Mov}(p)\}$. Since Π is finite, A is finite, and $b \notin A$ (each a_p avoids $\text{Mov}(g)$, while $b \in \text{Mov}(g)$).

Build the separating instance: let $I^* = \{b\} \cup A$, take $D = I^*$ and $f = \text{id}$, so $I = I^*$; take the suite $T = A$, so $O = A$. Then:

Every non-equivalent Π -mutant is killed. Let $p \in \Pi$ be non-equivalent, i.e. $\text{Mov}(p) \cap I \neq \emptyset$. If $b \in \text{Mov}(p)$, then $a_p \in \text{Mov}(p) \cap A = \text{Mov}(p) \cap O$, so p is killed. If $b \notin \text{Mov}(p)$, then $\text{Mov}(p) \cap I = \text{Mov}(p) \cap (\{b\} \cup A) = \text{Mov}(p) \cap A \neq \emptyset$, so p is killed. Hence $\text{score}_\Pi(f, T) = 1$.

$g \circ f$ survives. $b \in \text{Mov}(g) \cap I$, so $g \circ f$ is non-equivalent; but $\text{Mov}(g) \cap O = \text{Mov}(g) \cap A = \emptyset$, since every a_p avoids $\text{Mov}(g)$. So T does not kill $g \circ f$, and $\text{score}_\Gamma(f, T) < 1$. $\square$

Finiteness of Π is used only to keep A finite, so that the separating program has a finite reachable set. The general case weakens the containment to an avoidance condition.

Theorem 4.3 (footprint characterisation, general Π). *For arbitrary Π and Γ , Π is adequacy-complete for Γ iff*

$$\forall g \in \Gamma, \forall b \in \text{Mov}(g), \forall F \subseteq R \setminus \text{Mov}(g) \text{ finite}, \exists p \in \Pi : b \in \text{Mov}(p) \wedge \text{Mov}(p) \cap F = \emptyset.$$

Proof. ($\Leftarrow$) Assume the condition and fix (f, T) with $\text{score}_\Pi(f, T) = 1$ and non-equivalent $g \in \Gamma$. Pick $b \in \text{Mov}(g) \cap I$. Let $F = O \setminus \text{Mov}(g)$, which is finite (as O is) and disjoint from $\text{Mov}(g)$. Obtain $p \in \Pi$ with $b \in \text{Mov}(p)$ and $\text{Mov}(p) \cap F = \emptyset$. Since $b \in \text{Mov}(p) \cap I$, $p \circ f$ is non-equivalent, hence killed: there is

$c \in O \cap \text{Mov}(p)$. Since $\text{Mov}(p)$ avoids $F = O \setminus \text{Mov}(g)$ and $c \in O$, we get $c \in \text{Mov}(g)$. So $\text{Mov}(g) \cap O \neq \emptyset$ and T kills $g \circ f$.

($\Rightarrow$) By contraposition. Suppose the condition fails: there are $g \in \Gamma$, $b \in \text{Mov}(g)$, and a finite $F \subseteq R \setminus \text{Mov}(g)$ such that every $p \in \Pi$ with $b \in \text{Mov}(p)$ meets F . Take $I = \{b\} \cup F$, $D = I$, $f = \text{id}$, $T = F$, so $O = F$. If $p \in \Pi$ is non-equivalent and $b \in \text{Mov}(p)$, then $\text{Mov}(p) \cap F \neq \emptyset$ by assumption, so p is killed; if $b \notin \text{Mov}(p)$, then $\text{Mov}(p) \cap I = \text{Mov}(p) \cap F \neq \emptyset$, so p is killed. Hence $\text{score}_\Pi(f, T) = 1$. But $b \in \text{Mov}(g) \cap I$ while $\text{Mov}(g) \cap O = \text{Mov}(g) \cap F = \emptyset$, so $g \circ f$ is a non-equivalent survivor. $\square$

Corollary 4.4. *Theorem 4.2 is the case of Theorem 4.3 for finite Π .*

Proof. If the condition of Theorem 4.2 holds, then for any finite F disjoint from $\text{Mov}(g)$ the witness p with $\text{Mov}(p) \subseteq \text{Mov}(g)$ satisfies $\text{Mov}(p) \cap F = \emptyset$. Conversely, suppose the condition of Theorem 4.3 holds and Π is finite. If no p had $b \in \text{Mov}(p) \subseteq \text{Mov}(g)$, then each p with $b \in \text{Mov}(p)$ would have an escape point $a_p \notin \text{Mov}(g)$; the set A of these is finite and disjoint from $\text{Mov}(g)$, so applying Theorem 4.3 with $F = A$ yields a p with $b \in \text{Mov}(p)$ and $\text{Mov}(p) \cap A = \emptyset$, contradicting $a_p \in \text{Mov}(p) \cap A$. $\square$

Theorem 4.3 matters because several natural families are infinite—for instance the full value-guard family on an infinite return type (Section 5), which is absolutely complete but not finite, and therefore outside the reach of Theorem 4.2.

Remark 4.5 (composition is the wrong closure). Theorem 4.2 dissolves the temptation to define completeness by *generation under composition*. Killing the operators of Π does not kill their composites. Take $R = \{0, 1, 2\}$, a program with sole reachable and observed value 0, and operators a, b with $a(0) = 1$, $b(0) = 2$, $b(1) = 0$. Both $a \circ f$ and $b \circ f$ are killed at 0, yet $(b \circ a)(0) = b(1) = 0$, so $(b \circ a) \circ f$ is not killed—it is equivalent on this program. Composition acts precisely on the part of an operator that adequacy cannot see: $\text{Mov}(b \circ a)$ can be strictly smaller than both $\text{Mov}(a)$ and $\text{Mov}(b)$. So “ Π generates the full transformation monoid” says nothing about what a score of 1 certifies. The governing order is footprint containment, not compositional generation.

5 Absolute Completeness: Value Guards, the Ceiling, and the Dichotomy

Corollary 5.1 (the value-guard basis). *Let $|R| = n$. A family Π is absolutely complete iff it contains, for every $r \in R$, an operator whose footprint is the singleton $\{r\}$ —a value guard γ_r , readable as “if the result is r , return something else.” The minimal absolutely complete family is $\{\gamma_r : r \in R\}$, of size exactly n .*

Proof. Absolute completeness is completeness for $\Gamma = \mathcal{T}(R)$, whose footprints are all of 2^R (Proposition 3.5). Apply Theorem 4.2 to the singleton target footprints $\{b\}$: completeness forces, for each b , some $p \in \Pi$ with $b \in \text{Mov}(p) \subseteq \{b\}$,

i.e. $\text{Mov}(p) = \{b\}$. Conversely, given a guard for every value, any target footprint S and any $b \in S$ admit γ_b with $b \in \{b\} \subseteq S$, so Theorem 4.2’s condition holds. Minimality: each of the n singleton targets forces a distinct guard. $\square$

Corollary 5.2 (the finite/infinite dichotomy). *A finite operator family can be absolutely complete only when R is finite. Equivalently: for an infinite return type—*int*, *String*, arbitrary objects—no finite output-operator family is absolutely complete.*

Proof. By Corollary 5.1 absolute completeness requires a distinct singleton footprint $\{r\}$ for every $r \in R$; the map $r \mapsto \{r\}$ is injective, so a complete family has at least $|R|$ members. $\square$

Corollary 5.2 bounds the value-guard story to finite return types—Booleans, enums, small tagged unions. On the return types of most real programs the absolute notion is unavailable, and the meaningful question is *relative* completeness against a chosen target class (Section 6). That is where a certificate a tool can actually offer lives.

Corollary 5.3 (extreme mutation is optimal on Booleans, deficient beyond). *A Descartes-style constant operator `return c` has footprint $R \setminus \{c\}$. The constant family $\{\text{return } c : c \in R\}$ is absolutely complete iff $n = 2$.*

Proof. By Corollary 5.1, absolute completeness requires a singleton footprint for each value; $|R \setminus \{c\}| = 1$ iff $n = 2$. For $n = 2$, `return true` has footprint $\{\text{false}\}$ and `return false` has footprint $\{\text{true}\}$ —precisely the two value guards, so extreme mutation *is* the minimal complete family on Booleans. For $n \geq 3$ every constant footprint has size at least 2, so no constant is a value guard. $\square$

Example 5.4 (a fault the constants miss). Let $R = \{a, b, c\}$, a program reaching all three, and a suite observing only a and b (so $O = \{a, b\}$, $I = \{a, b, c\}$). All three constant mutants are killed: `return a` moves $\{b, c\}$, observed at b ; `return b` moves $\{a, c\}$, observed at a ; `return c` moves $\{a, b\}$, observed at a . The constant score is 1. But the fault “returns b where it should return c ”—an operator with footprint $\{c\}$ —is non-equivalent ($c \in I$) and unkillable ($\{c\} \cap O = \emptyset$). A perfect extreme-mutation score certifies nothing about the c -outputs the suite never observed.

Theorem 5.5 (the ceiling). *For $|R| \geq 2$ and any program f and suite T :*

$$\text{score}_{\mathcal{T}(R)}(f, T) = 1 \iff I \subseteq O \iff O = I,$$

i.e. an absolute mutation score of 1 holds exactly when the suite exercises every reachable output.

Proof. ($\Leftarrow$) If $O = I$, every non-equivalent g has $\text{Mov}(g) \cap I \neq \emptyset$, hence $\text{Mov}(g) \cap O \neq \emptyset$, so all die. ($\Rightarrow$) If $I \not\subseteq O$, pick $b \in I \setminus O$ and the guard γ_b with footprint $\{b\}$: it is non-equivalent ($b \in I$) and unkillable ($\{b\} \cap O = \emptyset$), so the score is below 1. $\square$

Remark 5.6 (the ceiling, read operationally). Theorem 5.5 gives the operational content of output mutation: a full score certifies *output coverage*—the dynamic property that every value the program can return is observed by the suite—and the value guards are the cheapest family making that certificate exact. It certifies nothing about faults that separate two inputs mapped to the same output: if $f(x_1) = f(x_2)$ then $p(f(x_1)) = p(f(x_2))$ for every output operator p , so no output mutant distinguishes x_1 from x_2 . Input-separating faults are structurally outside the reach of the technique.

6 Relative Completeness and Its Decidability

Fix a finite Π and a finitely presented target Γ . By Theorem 4.2, deciding completeness is deciding a footprint-containment condition, and its complexity is exactly that of the footprint representation.

Proposition 6.1 (the exact reach of a partial guard family). *The value guards $\{\gamma_{r_1}, \dots, \gamma_{r_k}\}$ are adequacy-complete for exactly $\Gamma_{\max} = \{g : \text{Mov}(g) \subseteq \{r_1, \dots, r_k\}\}$.*

Proof. By Theorem 4.2 the guards are complete for g iff every $b \in \text{Mov}(g)$ has a guard γ_{r_i} with $b \in \{r_i\} \subseteq \text{Mov}(g)$, i.e. $b \in \{r_1, \dots, r_k\}$. This holds for all $b \in \text{Mov}(g)$ iff $\text{Mov}(g) \subseteq \{r_1, \dots, r_k\}$. $\square$

Proposition 6.1 is the practically usable form of the theory. A partial guard family yields an explicit, checkable certificate: *every guarded value the program can produce is exercised by the suite, hence any fault confined to the guarded values is caught*. The guarded set is a parameter the tool chooses and reports.

Theorem 6.2 (decidability spectrum). *Deciding whether a finite Π is adequacy-complete for a finite Γ reduces to finitely many footprint-containment tests $\text{Mov}(p) \subseteq \text{Mov}(g)$ and membership tests $b \in \text{Mov}(p)$. Hence:*

1. **Finite return type (value tables).** *Footprints are explicit subsets of an n -element set; each test is a linear scan, so completeness is decidable in polynomial time.*
2. **Semilinear footprints** ($R = \mathbb{Z}$, Mov Presburger-definable). *Containment of Presburger-definable sets is decidable, so completeness is decidable.*
3. **Total computable operators.** *Each footprint $\text{Mov}(p)$ is a decidable set, but deciding containment—hence completeness—is undecidable.*

Proof. The reduction is Theorem 4.2, a finite conjunction over $g \in \Gamma$ and $b \in \text{Mov}(g)$, existentially quantifying $p \in \Pi$. (1) Tables give constant-time membership and linear-time containment. (2) Presburger arithmetic is decidable. (3) Reduce from non-halting: for a Turing machine M , the total computable operator $g_M(k) = k + 1$ if M halts within k steps and $g_M(k) = k$ otherwise has decidable footprint $\text{Mov}(g_M) = \{k : M \text{ halts within } k \text{ steps}\}$, and $\text{Mov}(g_M) \subseteq \text{Mov}(\text{id}) = \emptyset$ iff M never halts. $\square$

Remark 6.3 (the spectrum runs through set containment). The P / decidable / undecidable gradient here is that of subset containment for increasingly expressive set representations. No monoid- or group-theoretic machinery—word problems, submonoid membership, Cayley embeddings—appears, because adequacy never composes operators (Remark 4.5). It is the containment order, not a generation order, that governs decidability. An earlier formulation of operator completeness in terms of transformation-monoid generation leads to exactly that algebraic machinery, and to a different and, for adequacy purposes, incorrect answer; Remark 4.5 is the reason.

7 Higher-Order Mutants

The coupling-effect hypothesis [3] is the empirical claim that suites killing first-order mutants tend also to kill higher-order ones. For higher-order *output* mutants—compositions of output operators—no such implication holds as a theorem, and the footprint lens shows exactly why.

Proposition 7.1 (first-order completeness does not entail higher-order completeness). *There exist a family Π , a program f , and a suite T under which every non-equivalent first-order Π -mutant is killed while a non-equivalent higher-order mutant survives.*

Proof. Let $R = \{0, 1, 2\}$ and $\Pi = \{a\}$ with $a : 0 \mapsto 1, 1 \mapsto 0, 2 \mapsto 1$, so $\text{Mov}(a) = R$. Take a program f with reachable set $I = \{0, 2\}$ observed at $O = \{0\}$. The only first-order mutant $a \circ f$ is non-equivalent and killed, since $0 \in \text{Mov}(a) \cap O$. The higher-order mutant $a \circ a$ satisfies $a \circ a : 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0$, so $\text{Mov}(a \circ a) = \{2\}$: it meets I (at 2, hence non-equivalent) and misses O (hence unkilld). So the score is 1 against Π and below 1 against the closure of Π under composition. $\square$

By Proposition 3.4 the reason is exact: killing sees only $\text{Mov}(\cdot) \cap O$, and $\text{Mov}(a \circ a)$ can be strictly smaller than $\text{Mov}(a)$, removing precisely the value whose observation killed the first-order mutant.

Proposition 7.2 (the only positive statement is degenerate). *If Π is absolutely complete it is adequacy-complete for every target, in particular for its closure under composition.*

Proof. Corollary 5.1 gives Π a value guard for every value; Theorem 4.2's condition then holds for any target. $\square$

Proposition 7.2 is a corollary of the value-guard basis, not a coupling phenomenon: it holds because such a family already covers every footprint, so composition is irrelevant.

Remark 7.3 (what this does and does not say). By Theorem 4.2, higher-order-ness carries no special weight: a higher-order mutant is simply an operator with

a footprint, and the closure of Π under composition may contain footprints Π does not cover. We therefore claim only that coupling is not *entailed* for output mutation—not that it fails to occur. The coupling hypothesis is a statistical claim about typical faults, which a single counterexample does not refute. Program-text coupling is a separate question on which we make no claim.

8 Program Independence

Proposition 8.1 (program-independent subsumption). *For output operators p and g : every test that kills $p \circ f$ also kills $g \circ f$, for every program f , iff $\text{Mov}(p) \subseteq \text{Mov}(g)$.*

Proof. ($\Leftarrow$) A test x kills $p \circ f$ iff $f(x) \in \text{Mov}(p) \subseteq \text{Mov}(g)$, so x kills $g \circ f$. ($\Rightarrow$) If $b \in \text{Mov}(p) \setminus \text{Mov}(g)$, take f with b reachable and a suite observing b : it kills $p \circ f$ but not $g \circ f$. $\square$

So Theorem 4.2 reads, in the language of mutant subsumption [9,1]: Π is **adequacy-complete for Γ iff every target mutant is subsumed by some Π -mutant, uniformly over all programs**. It is the program-independent form of dominator theory.

Proposition 8.2 (the factoring that buys program independence). *For an output operator, the detection set factors through the footprint:*

$$\text{Det}(p \circ f) = \{x : p(f(x)) \neq f(x)\} = f^{-1}(\text{Mov}(p)).$$

So p enters detection only through the program-independent object $\text{Mov}(p)$: the same $\text{Mov}(p)$ governs detection across all programs f .

The contrast with program-text mutation is *not* that its detection is syntactic. The detection set $\text{Det}(m) = \{x : m(x) \neq f(x)\}$ of any mutant is semantic, depending only on the functions m and f . The contrast is that a program-text mutation *schema* carries no operator-only invariant: the mutant μ (source of f) depends on the *text* of f , so its detection is a function of the (schema, source) pair rather than of μ alone, and varies across source programs computing the same f . This is why the field’s completeness, sufficiency, and subsumption results are relative by construction (Section 2), and why Definition 4.1 is available for output mutation and not for program-text mutation.

9 The Minimum Certifying Suite

Theorem 9.1 (minimum certifying suite). *Fix an absolutely complete family and a program f with reachable set $I = f(D)$. A suite T achieves $\text{score}(f, T) = 1$ iff $O = f(T) = I$. If I is finite, the minimum certifying suite has size $\sigma(f) = |I| = |f(D)|$ (one test per reachable value). If I is infinite, no finite suite is certifying: a finite T has finite $O = f(T)$, so $O \neq I$.*

Proof. By Theorem 5.5 the score is 1 iff $O = I$. If I is finite, a suite with $O = I$ contains, for each $r \in I$, some x with $f(x) = r$, so $|T| \geq |I|$; choosing one such x per value gives $|T| = |I|$. If I is infinite, $O = f(T)$ is finite for finite T . $\square$

Remark 9.2 (sample complexity of certification). $\sigma(f)$ is the *teaching dimension* of the program against the operator family in the classical sense [5]: the minimum number of labelled examples—here, tests together with their observed outputs—required to distinguish f from every non-equivalent mutant the family can express. For output mutation it is exactly the number of reachable outputs. This is the exact-identification companion to the completeness result: Theorem 4.2 says *which* faults a family can certify against; Theorem 9.1 says *how many tests* the certification costs.

10 Beyond the Exact Oracle

The exact-oracle assumption (Section 3.4) is the model’s one load-bearing simplification, and the one that most constrains practical relevance, since the phenomenon motivating extreme mutation—pseudo-tested methods [11,15]—is a weak-oracle phenomenon by definition. This section shows the development relativises, and that the relativised model characterises pseudo-testedness.

Definition 10.1 (oracle). *An oracle is a map $\omega : R \rightarrow \Omega$ recording what the suite’s assertions actually observe about a returned value. The exact oracle is $\omega = \text{id}$. A suite that checks only a field, a hash, a type tag, or nothing at all is a non-injective ω ; a constant ω models a suite with no output assertions whatsoever.*

Definition 10.2 (oracle-relative killing). *T ω -kills $p \circ f$ iff $\omega(p(f(x))) \neq \omega(f(x))$ for some $x \in T$; $p \circ f$ is ω -equivalent iff $\omega(p(f(x))) = \omega(f(x))$ for all $x \in D$. The ω -score is 1 iff every non- ω -equivalent mutant is ω -killed.*

Taking ω -equivalence rather than true equivalence as the exclusion criterion matches tool behaviour: a mutant no assertion can ever distinguish is one the tool cannot kill and will report as equivalent.

Definition 10.3 (oracle-relative detection set). $\text{Det}_\omega(p) = \{r \in R : \omega(p(r)) \neq \omega(r)\}$.

Proposition 10.4 (the relativised reduction). *$p \circ f$ is ω -equivalent iff $\text{Det}_\omega(p) \cap I = \emptyset$; T ω -kills $p \circ f$ iff $\text{Det}_\omega(p) \cap O \neq \emptyset$. Moreover $\text{Det}_\omega(p) \subseteq \text{Mov}(p)$, with equality for injective ω .*

Proof. Identical to Proposition 3.4, reading $\omega \circ p$ against ω in place of p against the identity. Containment: if $p(r) = r$ then $\omega(p(r)) = \omega(r)$. $\square$

Corollary 10.5 (everything relativises). *Since $\text{Det}_\omega(p) \subseteq R$ and Proposition 10.4 has the form of Proposition 3.4, Theorems 4.2, 4.3, 5.5, 6.2, 9.1 and Propositions 6.1, 7.1, 7.2, 8.1 hold verbatim with Mov replaced throughout by Det_ω .*

The interesting content is where the two differ.

Proposition 10.6 (the ceiling is oracle-independent for any non-trivial oracle). *If ω is non-constant, then for every $b \in R$ there is an operator with $\text{Det}_\omega(p) = \{b\}$, and consequently the absolute ω -score is 1 iff $I \subseteq O$.*

Proof. Non-constant ω gives r_1, r_2 with $\omega(r_1) \neq \omega(r_2)$; for any b , at least one of them, say r' , has $\omega(r') \neq \omega(b)$. Set $p(b) = r'$ and $p = \text{id}$ elsewhere; then $\text{Det}_\omega(p) = \{b\}$. Apply the relativised Theorem 5.5. $\square$

So weakening the oracle does not lower the ceiling: output coverage remains exactly what a full absolute score certifies, provided the suite asserts *something*. What weakening the oracle changes is which *specific* families reach the ceiling—footprints shrink, so families lose discriminating power.

Proposition 10.7 (constants under a weak oracle). $\text{Det}_\omega(\text{return } c) = R \setminus \omega^{-1}(\omega(c))$. *The constant family is absolutely ω -complete iff $|R| = 2$ and ω separates the two values.*

Proof. $\omega(c) \neq \omega(r)$ iff r lies outside the ω -class of c . By the relativised Corollary 5.1, completeness requires each $\text{Det}_\omega(\text{return } c)$ to be a singleton for a suitable c per value b , i.e. $\omega^{-1}(\omega(c)) = R \setminus \{b\}$. For this to hold for every b , the ω -partition must be $\{\{b\}, R \setminus \{b\}\}$ for each b simultaneously, which forces $|R| = 2$ with ω injective on R . $\square$

Proposition 10.8 (pseudo-testedness, characterised). *Let f be covered by T (so $O \neq \emptyset$). The constant mutant $\text{return } c$ survives against T under oracle ω while being non- ω -equivalent iff*

$$O \subseteq \omega^{-1}(\omega(c)) \quad \text{and} \quad I \not\subseteq \omega^{-1}(\omega(c)),$$

i.e. every output the suite observes is oracle-indistinguishable from c , while some reachable output is not. Under the exact oracle this reduces to $O = \{c\}$.

Proof. Survival is $\text{Det}_\omega(\text{return } c) \cap O = \emptyset$, i.e. O avoids $R \setminus \omega^{-1}(\omega(c))$, i.e. $O \subseteq \omega^{-1}(\omega(c))$. Non- ω -equivalence is $\text{Det}_\omega \cap I \neq \emptyset$, i.e. $I \not\subseteq \omega^{-1}(\omega(c))$. Under $\omega = \text{id}$, $\omega^{-1}(\omega(c)) = \{c\}$. $\square$

This is the model's account of the empirical phenomenon: a method is pseudo-tested exactly when the suite's assertions collapse everything it observes into a single oracle class, so replacing the body by a constant in that class changes nothing the suite can see. It also explains why extreme mutation is a productive *detector* in practice even though the constant family is a weak *certifier* (Corollary 5.3): the two roles are different, and the gap $\text{Mov}(p) \setminus \text{Det}_\omega(p)$ is precisely the oracle weakness the technique surfaces.

Remark 10.9 (status). The oracle-relative detection set Det_ω and Propositions 10.6, 10.7, 10.8 are mechanised directly (Section 13, Table 2). What remains a paper proof is Corollary 10.5 itself: it is a substitution of one subset-valued invariant for another, so mechanising the *full* relativised theory is a matter of abstracting the Lean development over the detection function rather than reproving it, and we have not yet carried out that abstraction.

Table 1. Constant-mutant pseudo-testedness of covered functions/methods in five Python libraries, under the value (assertion) oracle and the crash-inclusive oracle. (†) `click` is shown but excluded from the pooled figure: 48 of its 60 sampled methods time out under in-process profiling, leaving too few measured to report.

Project	Covered w/ const	Value oracle	Crash oracle
<code>boltons</code>	27	63%	4%
<code>inflect</code>	33	52%	9%
<code>toolz</code>	46	33%	0%
<code>humanize</code>	16	19%	19%
<code>click</code> [†]	2	—	—
Pooled	122	43%	6%

11 Empirical: Pseudo-Testedness Is Oracle-Relative

Proposition 10.8 predicts that whether a covered method is *pseudo-tested* is a property not of the method but of the oracle: the constant mutant `return c` survives while remaining non- ω -equivalent exactly when every observed output is ω -indistinguishable from c . We test this directly, taking as two concrete oracles the standard distinction between an *assertion* kill—which pins the returned value, the exact oracle $\omega = \text{id}$ —and a *crash* (exception or timeout) kill, which proves only that the mutant ran differently and leaves the value unpinned: a weak, non-injective ω (Definition 10.2).

Setup. We ran the constant perturbation—`return 0` at each return site—against the *existing* test suites of five widely used Python libraries, over a seeded random sample of public functions and methods. A method enters the denominator only when a covering test actually executes the mutated return; an unexercised return is untested, not pseudo-tested. Matching the whole-body `return c` of extreme mutation [11,15], a multi-return method counts as pseudo-tested under an oracle iff *no* covered return is killed by that oracle. We report, per project, the fraction pseudo-tested under the value oracle (no assertion pins any return) and under the crash-inclusive oracle (no test—not even a crash—kills any return). The measurement uses the same engine that realises the paper’s operators; a run is discarded unless its baseline passes and its trace is complete.

Result. Across the 122 covered functions and methods of the four measurable projects, 43% (52/122) are pseudo-tested under the value oracle—squarely within the 1–46% that Vera-Pérez et al. measured across 21 Java projects [15]—but only 6% (7/122) under the crash-inclusive oracle. The 37-point gap is precisely the population of constant returns that a *traditional* mutation score counts as killed—a downstream crash caught them—while the returned *value* is unpinned. This is the phenomenon Section 10 formalises, measured directly: the same code is roughly seven times more pseudo-tested once crash detection is discounted, so pseudo-testedness is a property of the oracle and not of the suite

alone, confirming Proposition 10.8. The effect is uneven and interpretable: in `boltons` and `toolz` the surviving constants are almost all crash-killed and value-unpinned (63% vs. 4%; 33% vs. 0%), whereas `humanize`’s string formatters, collapsed to a constant, are either value-asserted or fully survive (19% under either oracle).

Threats to validity. The operator fixes $c = 0$; a family over several constants would only raise the rate, so the reported figures are lower bounds on pseudo-testedness. The sample is modest (242 probes) and, once class methods are included, skews toward accessors, which are less value-asserted than free functions—`boltons` at 63% sits above the Java ceiling for this reason, so we read the oracle *gap*, not the absolute rate, as the finding. `click` reflects a harness rather than a coverage limitation: its command/context objects do not profile in process. None of these affects the qualitative result, which is the ratio between the two oracles on identical code.

12 Consequences for Mutation-Based Tools

The results delimit as much as they prescribe. For a tool that certifies code against an output or extreme mutation family:

- **Report a passing score as output coverage, and only that.** By Theorem 5.5 an absolute score of 1 says “every value the program can return was observed by the suite.” Do not read it as evidence about faults that separate inputs mapped to the same output (Remark 5.6), nor about faults a weaker oracle would miss (Section 10).
- **Note that, for the absolutely complete family, output mutation adds no information over coverage.** A value-guard score of 1 is *equivalent* to output coverage, so a tool gains nothing by running the guard mutants rather than measuring output coverage directly. The value of Theorem 5.5 is that it says what the score means, not that it recommends computing it.
- **Where the certificate is partial, state it explicitly** (Proposition 6.1). On a large or infinite return type absolute completeness is unavailable (Corollary 5.2), so the useful output is relative: choose a target class Γ , compute the footprint basis Theorem 4.2 requires for it, and report “*complete against Π , which is adequacy-complete for Γ ; therefore this suite certifies property C* ” in place of an uninterpreted percentage. For a partial guard family the certificate reads “any fault confined to the guarded values is caught,” and the guarded set should be named.
- **Prefer value guards to constants past Booleans** (Corollary 5.3), if running output mutants at all. Constants are complete only for two-valued return types; beyond that they leave the surviving faults of Example 5.4.
- **Do not rely on coupling for output mutation** (Proposition 7.1). Killing first-order output mutants does not entail killing their higher-order compositions; if those are in scope, a tool must generate them.
- **Report the certifying-suite size** $\sigma(f) = |f(D)|$ (Theorem 9.1) as the cost and the content of the guarantee.

Table 2. Status of each result. “Mechanised” means a Lean theorem with no `sorry` and a clean axiom audit.

Result	Lean name	Status
Prop. 3.4	<code>progScore_iff_scoreAt</code>	mechanised
Prop. 3.5	<code>mov_surjective</code> , <code>mov_image_univ</code>	mechanised
Thm. 4.2	<code>footprint_characterization</code>	mechanised
Thm. 4.2 (programs)	<code>progComplete_characterization</code>	mechanised
Thm. 4.3	<code>footprint_characterization_general</code>	mechanised
Cor. 5.1	<code>absolute_iff_guards</code>	mechanised
Cor. 5.2	<code>progComplete_univ_infinite</code>	mechanised
Cor. 5.3	<code>constants_iff_card_two</code>	mechanised
Thm. 5.5	<code>ceiling</code>	mechanised
Prop. 6.1	—	paper proof
Thm. 6.2	—	paper proof (reduction is Thm. 4.2)
Prop. 7.1	<code>coupling_fails</code>	mechanised
Prop. 7.2	<code>coupling</code> , <code>progComplete_coupling</code>	mechanised
Prop. 8.1	<code>subsumes_iff_subset</code>	mechanised
Prop. 8.2	<code>det_factors</code>	mechanised (rfl)
Thm. 9.1 (finite)	<code>certify_lb</code> , <code>certify_ub</code>	mechanised (observed-set core)
Thm. 9.1 (infinite)	<code>certify_infinite</code>	mechanised
Bridge	<code>realizable</code> , <code>progComplete_iff_complete</code>	mechanised
Prop. 10.6 (guard core)	<code>detW_singleton_of_nonconstant</code>	mechanised
Prop. 10.7 (identity)	<code>detW_const</code>	mechanised (rfl)
Prop. 10.8	<code>pseudo_tested_iff</code>	mechanised
Cor. 10.5 (relativisation)	—	paper proof

13 Mechanisation

The development is machine-checked in Lean 4 against Mathlib. The artifact defines completeness *directly over programs and finite suites*—`ProgScore` and `ProgComplete` range over $f : D \rightarrow R$ and $T : \text{Finset } D$ —and then bridges to the footprint level: `progScore_iff_scoreAt` (Proposition 3.4) reduces a program’s score to the footprint-level `ScoreAt` at the reachable set `range f` and observed set `f " T`; `realizable` shows every reachable/observed pair (I, O) with $O \subseteq I$ is realised by an actual program; `progComplete_iff_complete` glues these together; and `progComplete_characterization` restates the main theorem over programs.

Every mechanised theorem is `#print axioms-clean`, depending on exactly `[propext, Classical.choice, Quot.sound]` (`det_factors` and `detW_const` on none), with no `sorryAx`.

Two caveats stated plainly. First, Theorem 9.1 is mechanised at the level of the *observed set*: `certify_lb` and `certify_ub` establish that the minimum $|O|$ is $|I|$ for finite I , and the identity $|T| = |O|$ for a minimal suite is the immediate preimage choice, done by hand. Second, Theorem 6.2’s spectrum is

a meta-statement about representations; its load-bearing reduction to footprint containment *is* the mechanised Theorem 4.2, but the complexity and computability claims are paper proofs.

Artifact. The Lean development accompanies this paper. To reproduce: place the file in a Lean 4 project with Mathlib, run `lake exe cache get`, and check e.g. `#print axioms progComplete_characterization`.

14 Threats to Validity and Scope

The exact oracle. Sections 4–9 assume the suite observes the full return value. Section 10 removes this assumption and shows the theory relativises, with the ceiling unchanged for any non-constant oracle and pseudo-testedness characterised (Proposition 10.8). The relativised results are paper proofs.

Purity and totality. Denotations are total functions. Void methods, exceptions, non-termination, and side effects are outside the model. Programs with these features can be brought inside by enlarging the return type with the corresponding observations, but we do not develop the encoding, and the enlarged type is typically infinite, which by Corollary 5.2 puts absolute completeness out of reach and moves the question to relative completeness.

Uniform post-composition. An output operator perturbs every returned value identically. A statement-level mutation of one `return` site among several is not an output operator, and the theory does not cover it.

Applicability of the ceiling. Theorem 5.5 shows a full absolute score is equivalent to output coverage. This is a statement about what the score *means*; it is not an empirical claim about how often suites achieve it, nor about how well output coverage correlates with fault detection. Those are separate, empirical questions.

Scope of the negative results. Proposition 7.1 shows coupling is not *entailed*. It does not show that coupling fails in practice, and it says nothing about program-text mutation.

15 Conclusion and Future Work

Mutation testing is ordinarily used heuristically: a high score is read as evidence of suite adequacy, justified by the coupling effect and the competent-programmer hypothesis, which are empirical. For one operator class we take a different stance—measuring a family not by the faults it can *express* but by what killing its mutants *certifies*, uniformly over programs and suites—and characterise the score exactly.

The characterisation partitions the content of a passing score into three determinate parts: what it **certifies** (under an absolutely complete family, precisely

output coverage, Theorem 5.5); what it **provably cannot** certify (any fault distinguishing two inputs the program maps to the same output, Remark 5.6); and where the completeness question is **undecidable** (relative completeness against a presented target, off finite return types, Section 6). For this class a mutation score is an object with a characterised semantics rather than a quantity trusted empirically.

Four directions remain open.

1. **Other operator classes.** Proposition 8.2 identifies precisely what would let another class admit the same treatment: a program-independent detection invariant analogous to the footprint. Characterising the largest operator class whose detection factors through such an invariant would say how far this style of result can reach.
2. **The weak-oracle theory.** Section 10 shows the development relativises and characterises pseudo-testedness. Mechanising the relativised theory—abstracting the Lean development over the detection function—and studying which oracles arise from real assertion styles is the natural next step, and the one with the most contact with practice.
3. **Cost-optimal partial families.** Given a target class Γ and a per-operator cost, the minimum-cost Π adequacy-complete for Γ is a weighted set-cover over footprints; Theorem 4.2 makes the feasibility predicate explicit.
4. **Decidable relative fragments.** Theorem 6.2 gives polynomial time for explicit finite footprints and decidability for Presburger-definable ones. A systematic account of the boundary—which footprint representations admit decidable containment—is open.

References

1. Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: ICST (2014)
2. Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. *Acta Informatica* **18**(1), 31–45 (1982)
3. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *IEEE Computer* **11**(4), 34–41 (1978)
4. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: CADE (2021)
5. Goldman, S.A., Kearns, M.J.: On the complexity of teaching. *J. Comput. Syst. Sci.* **50**(1), 20–31 (1995)
6. Hamlet, R.G.: Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* **SE-3**(4), 279–290 (1977)
7. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)
8. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: FSE (2014)
9. Kurtz, B., Ammann, P., Offutt, J.: Static analysis of mutant subsumption. In: ICST Workshops (Mutation), pp. 1–10 (2015). <https://doi.org/10.1109/ICSTW.2015.7107454>

10. The mathlib Community: The Lean mathematical library. In: CPP (2020)
11. Niedermayr, R., Juergens, E., Wagner, S.: Will my tests tell me if I break this code? In: CSED@ICSE, pp. 23–29 (2016). <https://doi.org/10.1145/2896941.2896944>
12. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. **1**(1), 5–20 (1992)
13. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Trans. Softw. Eng. Methodol. **5**(2), 99–118 (1996)
14. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: an analysis and survey. Advances in Computers **112**, 275–378 (2019)
15. Vera-Pérez, O.L., Danglot, B., Monperrus, M., Baudry, B.: A comprehensive study of pseudo-tested methods. Empirical Software Engineering **23**(3), 1908–1935 (2018)