Praktische Mutationstests im Maßstab

**Zusammenfassung** - Die Mutationsanalyse bewertet die Angemessenheit einer Testsuite, indem sie ihre Fähigkeit misst, kleine künstliche Fehler zu erkennen, die systematisch in das getestete Programm eingepflanzt werden. Die Mutationsanalyse gilt als eines der stärksten Testadäquanzkriterien. Mutationstests bauen auf der Mutationsanalyse auf und sind eine Testtechnik, die Mutanten als Testziele verwendet, um eine Testsuite zu erstellen oder zu verbessern. Mutationstests gelten seit langem als unlösbar, da die schiere Anzahl der Mutanten, die erstellt werden können, ein unüberwindbares Problem darstellt - sowohl in Bezug auf den menschlichen als auch den Rechenaufwand. Dies hat die Einführung von Mutationstests als Industriestandard behindert. Zum Beispiel hat Google eine Codebasis von zwei Milliarden Codezeilen und mehr als 150.000.000 Tests werden täglich ausgeführt. Der traditionelle Ansatz für Mutationstests skaliert nicht auf eine solche Umgebung; selbst bestehende Lösungen zur Beschleunigung der Mutationsanalyse reichen nicht aus, um sie in einem solchen Maßstab rechnerisch realisierbar zu machen.

Um diese Herausforderungen anzugehen, stellt dieses Papier einen skalierbaren Ansatz für Mutationstests vor, der auf den folgenden Hauptideen basiert:

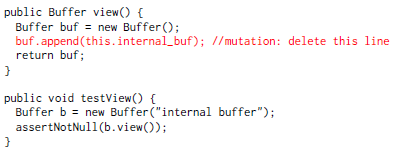
(1) Mutationstests werden inkrementell durchgeführt und mutieren nur geänderten Code während der Codeüberprüfung und nicht die gesamte Codebasis;

(2) Mutanten werden gefiltert, wobei Mutanten entfernt werden, die für Entwickler wahrscheinlich irrelevant sind, und die Anzahl der Mutanten pro Zeile und pro Code-Review-Prozess begrenzt wird; (3) Mutanten werden basierend auf der historischen Leistung von Mutationsoperatoren ausgewählt, wodurch irrelevante Mutanten weiter eliminiert und die Mutantenqualität verbessert wird. Dieses Papier validiert den vorgeschlagenen Ansatz empirisch, indem es seine Wirksamkeit in einer Code-Review-basierten Umgebung analysiert, die von mehr als 24.000 Entwicklern in mehr als 1.000 Projekten verwendet wird. Die Ergebnisse zeigen, dass der vorgeschlagene Ansatz um Größenordnungen weniger Mutanten produziert und dass kontextbasierte Mutantenfilterung und Selektion die Qualität und Umsetzbarkeit der Mutanten verbessern. Insgesamt stellt der vorgeschlagene Ansatz ein Mutationstest-Framework dar, das sich nahtlos in den Software-Entwicklungsworkflow integriert und auf industrielle Einstellungen jeder Größe anwendbar ist.

**Indexbegriffe** - Mutationstest, Codeabdeckung, Testwirksamkeit

**1 EINLEITUNG**

Softwaretests sind die vorherrschende Technik zur Sicherstellung der Softwarequalität, und es gibt verschiedene Ansätze zur Bewertung der Wirksamkeit der Testsuite (d. H. Die Fähigkeit einer Testsuite, Softwarefehler zu erkennen). Ein gängiger Ansatz ist die Codeabdeckung, die bei Google [1] weit verbreitet ist und den Grad misst, in dem eine Testsuite ein Programm ausübt. Die Codeabdeckung ist intuitiv, billig zu berechnen und wird von Tools für kommerzielle Zwecke gut unterstützt. Die Codeabdeckung allein ist jedoch unzureichend und kann ein falsches Gefühl der Wirksamkeit vermitteln, insbesondere wenn Programmanweisungen abgedeckt sind, aber ihr erwartetes Ergebnis nicht geltend gemacht wird [2], [3]. Ein alternativer Ansatz, der diese Einschränkung adressiert, ist die Mutationsanalyse, die künstliche Fehler, sogenannte Mutanten, systematisch in ein Programm einordnet und die Fähigkeit einer Testsuite misst, sie zu erkennen [4]. Mutationsanalyseadressen gelten weithin als der beste Ansatz zur Bewertung der Wirksamkeit von Testsuiten [5], [6], [7]. Mutationstests sind ein iterativer Testansatz, der auf der Mutationsanalyse aufbaut und unentdeckte Mutanten als konkrete Testziele verwendet, um den Testprozess zu leiten. Betrachten Sie als konkretes Beispiel die folgende vollständig abgedeckte, aber schwach getestete Funktionsansicht:



Der Test übt die Funktion aus, behauptet aber nicht auf seine Auswirkungen auf den zurückgegebenen Puffer. In diesem Fall übertrifft die Mutationsanalyse die Codeabdeckung: Obwohl die Zeile, die einige Inhalte an buf anhängt, abgedeckt ist, wird ein Entwickler nicht darüber informiert, dass kein Test seine Auswirkungen überprüft. Die im Codebeispiel hervorgehobene Statement-Deletion-Mutation weist explizit auf diese Testschwäche hin: Der Test scheitert nicht beim Einfügen dieses künstlichen Defekts. Google ist stets bestrebt, die Testqualität zu verbessern, und hat sich daher entschlossen, Mutationstests einzuführen und einzusetzen, um deren Wirksamkeit zu bewerten. Der Umfang der Codebasis von Google mit etwa 2 Milliarden Codezeilen machte den traditionellen Ansatz für Mutationstests jedoch unmöglich: Mehr als 150.000.000 Testausführungen pro Tag sind Gatekeeper für 40.000 Änderungseinreichungen in diese Codebasis, wodurch sichergestellt wird, dass 14.000 kontinuierliche Integrationen täglich gesund bleiben [8], [9]. Erstens erzeugt die systematische Mutierung der gesamten Codebasis oder sogar einzelner Projekte eine beträchtliche Anzahl von Mutanten, von denen jede möglicherweise die Durchführung vieler Tests erfordert. Zweitens wäre weder das traditionell berechnete Mutanten-Detektionsverhältnis, das die Wirksamkeit der Testsuite quantifiziert, noch das einfache Zeigen aller Mutanten, die dem Nachweis entgangen sind, an einen Entwickler umsetzbar. Angesichts der Tatsache, dass die Auflösung einer einzelnen Mutante mehrere Minuten dauert [10], [11], wäre der erforderliche Entwickleraufwand zur Auflösung aller unentdeckten Mutanten selbst in kleinem Maßstab unerschwinglich teuer.

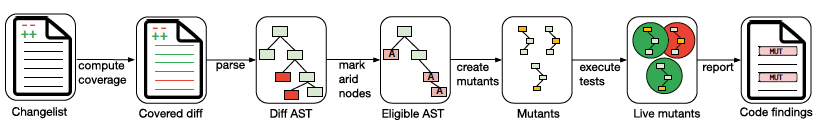


Abb. 1: Der Mutationstestdienst. Für eine bestimmte Änderungsliste wird die Zeilenabdeckung berechnet und der Code in eine AST geparst. Für AST-Knoten, die abgedeckte Linien überspannen, werden aride Knoten mit der Heuristik der ariden Knotenerkennung markiert, und nur nicht-aride (berechtigte) Knoten werden mutiert. Die erzeugten Mutanten werden getestet, und überlebende Mutanten werden als Codebefunde gemeldet.

Erschwerend kommt hinzu, dass selbst bei der Anwendung von Probenahmetechniken, um die Anzahl der Mutanten erheblich zu reduzieren, die Entwickler bei Google zunächst 85% der gemeldeten Mutanten als unproduktiv eingestuft haben. Eine unproduktive Mutante ist entweder trivial äquivalent zum ursprünglichen Programm oder sie ist nachweisbar, aber das Hinzufügen eines Tests dafür würde die Testsuite nicht verbessern [11].

Z.B. Mutieren der Anfangskapazität einer Java-Sammlung (z.B. neue ArrayList (64) 7! new ArrayList (16)) erzeugt eine unproduktive Mutante. Während es möglich ist, einen Test zu schreiben, der sich auf die Sammelkapazität oder die erwarteten Speicherzuweisungen bezieht, ist es unproduktiv, dies zu tun. In der Tat ist es denkbar, dass diese Tests, wenn sie geschrieben und hinzugefügt werden, sogar negative Auswirkungen haben würden, weil ihre Art des Änderungsdetektors (insbesondere das Testen der aktuellen Implementierung anstelle der Spezifikation) gegen die Best Practices des Testens verstößt und spröde Tests und Fehlalarme verursacht. Angesichts der zwei großen Herausforderungen beim Einsatz von Mutationstests - den Rechenkosten der Mutationsanalyse und der Tatsache, dass die meisten Mutanten unproduktiv sind - haben wir einen Mutationstestansatz entwickelt, der skalierbar und nutzbar ist und auf drei zentralen Ideen basiert:

1) Unser Ansatz führt Mutationstests für Codeänderungen durch: Er berücksichtigt nur geänderte Codezeilen und meldet Mutanten während der Codeüberprüfung (Abschnitt 2, basierend auf unserer früheren Arbeit [12]). Dies reduziert die Anzahl der Zeilen, in denen Mutanten erzeugt werden, erheblich und entspricht der Arbeitseinheit eines Entwicklers, für die zusätzliche Tests wünschenswert sind.

2) Unser Ansatz verwendet transitive Mutantenunterdrückung: Es verwendet Heuristiken basierend auf Entwickler-Feedback (Abschnitt 3, basierend auf unserer früheren Arbeit [12]). Das Feedback von mehr als 20.000 Entwicklern zu Tausenden von Mutanten über sechs Jahre ermöglichte es uns, Heuristiken zur Mutantenunterdrückung zu entwickeln, die das Verhältnis unproduktiver Mutanten von 85% auf 11% reduzieren.

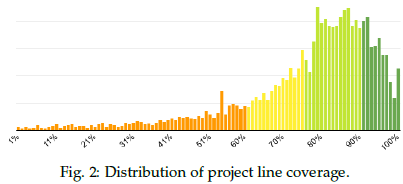
3) Unser Ansatz verwendet probabilistische, gezielte Mutantenselektion: Es berichtet über eine begrenzte Anzahl von Mutanten basierend auf der historischen Mutantenleistung, wodurch unproduktive Mutanten weiter vermieden werden (Abschnitt 4). Unsere Bewertung des vorgeschlagenen Ansatzes umfasste 760.000 Codeänderungen und 2 Millionen Mutanten, die während der Codeüberprüfung gemeldet wurden, von insgesamt fast 17 Millionen generierten Mutanten (Abschnitt 5). Die Ergebnisse zeigen, dass unser Ansatz Mutationstests machbar und umsetzbar macht - auch für industrielle Softwareentwicklungsumgebungen.

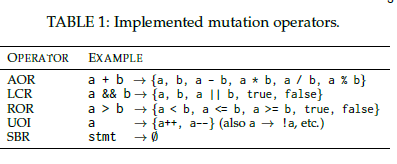
**2 MUTATIONSTESTS BEI GOOGLE**

Mutationstests bei Google stehen vor großen Herausforderungen, sowohl in Bezug auf die Rechenzeit als auch auf die Integration in den Entwicklerworkflow. Obwohl bestehende Arbeiten an selektiven Mutationen und anderen Optimierungen [13] die Anzahl der Mutanten, die analysiert werden müssen, erheblich reduzieren können, bleibt es aufgrund seiner Größe unerschwinglich teuer, das Mutantdetektionsverhältnis für die gesamte Codebasis von Google zu berechnen. Es wäre sogar noch teurer, das Mutanten-Erkennungs-Verhältnis neu zu berechnen, z.B. täglich oder wöchentlich, und es ist unmöglich, es nach jedem Commit zu berechnen. Zusätzlich zu den Kosten für die Berechnung dieses Verhältnisses konnten wir keinen guten Weg finden, es den Entwicklern auf umsetzbare Weise zu melden: Es ist weder konkret noch umsetzbar und führt nicht zu Tests. Die Meldung einzelner Mutanten in großem Maßstab an Entwickler ist ebenfalls eine Herausforderung, insbesondere aufgrund unproduktiver Mutanten.

Um die Herausforderungen von Skala und unproduktiven Mutanten anzugehen, haben wir einen Mutationstestansatz entwickelt und implementiert, der sich vom traditionellen Ansatz unterscheidet, der in der Literatur beschrieben wird [14]. Für die Skalierbarkeit haben wir diff-basierte Mutationstests entwickelt und implementiert, die nur Mutanten für abgedeckte, geänderte Linien generieren und bewerten. Für die Produktivität haben wir einen Ansatz zur Mutantenunterdrückung und probabilistischen Mutantenselektion entwickelt und implementiert. Mutationstests bei Google beginnen, wenn ein Entwickler eine Codeänderung zur Codeüberprüfung sendet. Der Mutationstestprozess besteht aus vier übergeordneten Schritten: Code-Coverage-Analyse (Abschnitt 2.1), Mutantengenerierung (Abschnitt 2.2), Mutationsanalyse (Abschnitt 2.3) und Meldung überlebender Mutanten im Code-Review-Prozess (Abschnitt 2.4).

Abbildung 1 zeigt den Mutationstestdienst. (1) Es beginnt mit einer Änderungsliste, die zur Codeüberprüfung eingereicht wird. (2) Sobald codecoverage-Metadaten verfügbar sind, bestimmt es den Satz von Zeilen, die abgedeckt und in der Änderungsliste hinzugefügt oder geändert werden. (3) Es erstellt dann eine AST jeder betroffenen Datei und besucht jeden abgedeckten Knoten. (4) Es markiert dann aride Knoten (Knoten, die, wenn sie mutiert sind, unproduktive Mutanten erzeugen), basierend auf der Heuristik, die unter Verwendung von Entwickler-Feedback über die Produktivität von Mutanten im Laufe der Jahre akkumuliert wurde. Die Markierung von ariden Knoten erfolgt, bevor Mutanten erzeugt werden, und daher werden Mutanten in ariden Knoten überhaupt nicht erzeugt. (5) Die Mutagenese erzeugt dann Mutanten für geeignete Knoten (d.h. jeden Knoten, der nicht arid ist und der von mindestens einem Test abgedeckt wird). (6) Der Mutationstestdienst wertet dann die Mutanten gegen die bestehenden Tests aus und (7) meldet eine Untergruppe überlebender Mutanten als Codebefunde im Code-Review.





**2.1 Voraussetzungen: Änderungslisten und Abdeckung**

Eine Änderungsliste ist eine atomare Aktualisierung des Versionskontrollsystems und besteht aus einer Liste von Dateien, den an diesen Dateien auszuführenden Operationen und möglicherweise den zu ändernden oder hinzuzufügenden Dateiinhalten sowie Metadaten wie Änderungsbeschreibung, Autor usw.

Sobald ein Entwickler eine Änderungsliste an Peer-Entwickler zur Codeüberprüfung sendet, werden verschiedene statische und dynamische Analysen für diese Änderungsliste durchgeführt und die Ergebnisse werden dem Entwickler und den Prüfern gemeldet. Die Linienabdeckung ist eine solche Analyse: Während der Codeüberprüfung wird die Gesamt- und Delta-Codeabdeckung den Entwicklern gemeldet [1]. Die Gesamtcodeabdeckung ist das Verhältnis der Anzahl der Zeilen, die von den Tests in der Datei abgedeckt werden, zur Gesamtzahl der instrumentierten Zeilen in der Datei. Die Anzahl der instrumentierten Zeilen ist in der Regel kleiner als die Gesamtzahl der Zeilen, da Artefakte wie Kommentare oder reine Leerraumzeilen ausgeschlossen sind. Die Delta-Abdeckung ist das Verhältnis der Anzahl der abgedeckten hinzugefügten oder geänderten Zeilen zur Gesamtzahl der hinzugefügten oder geänderten Zeilen in der Änderungsliste. Abbildung 2 zeigt die Linienabdeckungsverteilung pro Projekt, was darauf hinweist, dass die Linienabdeckung der meisten Projekte zufriedenstellend ist.

Die Codeabdeckung ist eine Voraussetzung für die Durchführung einer Mutationsanalyse aufgrund der hohen Kosten für die Generierung und Bewertung von Mutanten in nicht abgedeckten Zeilen, die alle unvermeidlich überleben würden, da der Code nicht getestet wird. Sobald eine Abdeckung auf Zeilenebene für eine Änderungsliste verfügbar ist, wird die Mutagenese ausgelöst. Google verwendet Bazel als Build-System [15]. Build-Ziele listen explizit ihre Quellen und Abhängigkeiten auf und entsprechen einer beliebigen Anzahl von Testzielen, von denen jedes mehrere Tests beinhalten kann. Tests werden parallel durchgeführt. Unter Verwendung der expliziten Abhängigkeit und der Quellenauflistung liefert die Codecoverage-Analyse Informationen darüber, welches Testziel welche Zeilen im Quellcode abdeckt, wodurch Codezeilen mit einem Satz von Tests verknüpft werden, die sie abdecken. Die Abdeckung auf Leitungsebene wird verwendet, um den Satz von Tests zu bestimmen, die ausgeführt werden müssen, um eine Mutante zu töten. Dieser Ansatz wird auch in anderen Mutationstestwerkzeugen umgesetzt, einschließlich PIT [16] und Major [17], [18].

**2.2 Mutagenese**

Der Mutagenese-Dienst empfängt eine Anforderung, Punktmutationen zu erzeugen, d.h. Mutationen, die eine Mutante erzeugen, die sich von der ursprünglichen in einem AST-Knoten auf der angeforderten Zeile unterscheidet. Für jede unterstützte Programmiersprache akzeptiert ein spezieller Mutagenesedienst, der in der Lage ist, die AST einer Kompilierungseinheit in dieser Sprache zu navigieren, Punktmutationsanforderungen und Antworten mit potenziellen Mutanten. Die Mutationsoperatoren werden als AST-Besucher implementiert, ein Ansatz, der auch von anderen Mutationswerkzeugen verfolgt wird (z. B. [19]). Für jede Punktmutationsanforderung, d.h. eine (Datei; Linie) tupel, ein Mutationsoperator ausgewählt wird und eine Mutante in dieser Linie erzeugt wird, wenn dieser Mutationsoperator auf ihn anwendbar ist. Wenn keine Mutante vom Mutationsoperator generiert wird, wird ein anderer Operator ausgewählt und so weiter, bis entweder eine Mutante generiert wird oder alle Mutationsoperatoren ausprobiert wurden und keine Mutante generiert werden konnte. Es gibt zwei Mutationsoperatoren-Auswahlstrategien, zufällig und gezielt, die in Abschnitt 4 beschrieben werden. Der Mutationstestdienst generiert aus Gründen der Skalierbarkeit und basierend auf der Erkenntnis, dass die überwiegende Mehrheit der Mutanten für eine bestimmte Linie dasselbe Schicksal teilen - entweder alle oder keine von ihnen überleben die Analyse [20]. Dies bedeutet, dass, wenn eine für eine bestimmte Linie generierte Mutante die Mutationsanalyse nicht überlebt, keine zusätzlichen Mutanten für diese Linie generiert werden. Der Mutationstestdienst implementiert Mutagenese für 10 Programmiersprachen: C++, Java, Go, Python, TypeScript, JavaScript, Dart, SQL, Common Lisp und Kotlin. Für jede Sprache implementiert der Dienst fünf Mutationsoperatoren: AOR (Arithmetic Operator Replacement), LCR (Logical Connector Replacement), ROR (Relational Operator Replacement), UOI (Unary Operator Insertion) und SBR (Statement Block Removal). Diese Mutationsoperatoren wurden ursprünglich für Mothra [21] eingeführt, und Tabelle 1 gibt ein Beispiel für jeden. In Python werden unäre Inkremente und Dekremente durch einen binären Operator ersetzt, um aufgrund des Sprachdesigns den gleichen Effekt zu erzielen. Nach unserer Erfahrung schuf der ABS (Absolute value insertion) Mutationsoperator überwiegend unproduktive Mutanten, hauptsächlich weil er auf zeit- und zählbezogene Ausdrücke reagierte, die positiv und unsinnig sind, wenn sie negiert werden. Daher verwendet der Mutationstestdienst den ABS-Operator nicht. Beachten Sie, dass unsere Beobachtungen möglicherweise nicht allgemein gültig sind und eine Funktion des Stils und der Merkmale unserer Codebasis sein können.

**2.3 Mutationsanalyse**

Sobald die Mutagenese einen Satz von Mutanten für eine Änderungsliste erzeugt hat, wird ein temporärer Zustand des Versionskontrollsystems für jeden von ihnen basierend auf der ursprünglichen Änderungsliste vorbereitet, und dann werden Tests parallel für alle diese Zustände ausgeführt. Dies ermöglicht eine effiziente Interaktion und Zwischenspeicherung zwischen unserem Versionskontrollsystem und dem Build-System und bewertet Mutanten auf schnellstmögliche Weise.

Sobald die Ergebnisse der Mutationsanalyse verfügbar sind, wählt der Mutationstestdienst Mutanten aus dem Satz überlebender Mutanten aus und meldet sie. Wir begrenzen die Anzahl der gemeldeten Mutanten auf höchstens das 7-fache der Gesamtzahl der Dateien in einer Änderungsliste. Dies stellt sicher, dass der kognitive Aufwand für das Verständnis aller gemeldeten Mutanten nicht zu hoch ist, was ansonsten dazu führen könnte, dass Entwickler keine Mutationstests mehr verwenden. Wir haben empirisch 7 als angemessenen Kompromiss zwischen Testwirksamkeit und kognitiver Belastung ermittelt, indem wir Daten über die Jahre des Systembetriebs gesammelt haben. Schließlich meldet der Dienst ausgewählte überlebende Mutanten in der Code-Review-UI an den Autor und die Gutachter. Beachten Sie, dass der Mutationstestdienst aus Konsistenzgründen Mutanten in derselben Zeile wie zuvor auswählt und meldet, wenn ein Autor zusätzliche Tests hinzufügt oder die Änderungsliste anderweitig aktualisiert, was eine erneute Ausführung des Dienstes auslöst.

**2.4 Meldung von Mutanten im Code-Review-Prozess**

Die meisten Änderungen an Googles Codebasis, mit Ausnahme einer begrenzten

Anzahl der vollautomatischen Änderungen, werden von Entwicklern überprüft

bevor sie mit dem Quellbaum zusammengeführt werden. Potvin

und Levenberg [9] bieten einen umfassenden Überblick über

Das Entwicklungsökosystem von Google. Prüfer können Kommentare hinterlassen

auf den geänderten Code, der vom Autor aufgelöst werden muss.

Eine spezielle Art von Kommentar, der durch eine automatisierte

Analysator wird als Befund bezeichnet. Im Gegensatz zu vom Menschen erzeugten

Kommentare, Ergebnisse müssen nicht vom Autor gelöst werden

vor der Einreichung, es sei denn, ein menschlicher Rezensent kennzeichnet sie als

obligatorisch. Viele Analysatoren werden automatisch ausgeführt, wenn ein

Änderungsliste wird zur Überprüfung gesendet: Linter, Formatierer, statischer Code

und bauen Abhängigkeitsanalysatoren, etc. Die Mehrheit der Analysatoren

basieren auf der Tricorder-Code-Analyse-Plattform [22].

Der Mutationstestdienst meldet ausgewählte Mutanten

Entwickler während des Code-Review-Prozesses, der maximiert

die Chancen, dass diese von den Entwicklern berücksichtigt werden.

Die Anzahl der während des Codes angezeigten Kommentare

Überprüfung kann groß sein, so ist es wichtig, dass alle Werkzeuge produzieren

umsetzbare Erkenntnisse, die sofort vom

Entwickler. Meldung nicht umsetzbarer Befunde während des Codes

Review hat negative Auswirkungen auf den Autor und die Gutachter.

Wenn ein Befund (z. B. eine überlebende Mutante) nicht wahrgenommen wird

als nützlich, können Entwickler berichten, dass mit einem einzigen Klick auf

den Befund. Wenn einer der Prüfer einen Befund für

wichtig, sie können angeben, dass die Änderungsliste Autor

mit einem einzigen Klick. Abbildung 3 zeigt eine Beispielmutante

angezeigt in Kritik, Googles Code Review System [23],

einschließlich der Links „Bitte beheben“ und „Nicht nützlich“ im

unteren Ecken. Dieses Feedback ist für den Inhaber von

das System, das die Ergebnisse erstellt hat, so dass Qualitätsmetriken

verfolgt werden, und nicht umsetzbare Ergebnisse triagiert und idealerweise

zukünftig verhindert.

Um für den Autor und die Rezensenten von Nutzen zu sein, Code

Ergebnisse müssen umsetzbar sein und schnell berichtet werden, bevor

die Überprüfung abgeschlossen ist. Zu diesem Zweck wird der Mutationstest

Dienst führt Mutantenunterdrückung durch (Abschnitt 3), und es

probabilistisch wählt Mutanten auf der Grundlage ihrer historischen

Leistung des Mutationsoperators (Abschnitt 4).

**3 UNTERDRÜCKUNG UNPRODUKTIVER MUTANTEN**

Einige Teile des Codes sind weniger interessant als andere.

Meldung von lebenden Mutanten in uninteressanten Aussagen (z.

Protokollierungsanweisungen für Debugging-Zwecke) eine negative

Einfluss auf die kognitive Belastung und den Zeitaufwand für die Analyse von Mutanten.

Weil Entwickler das Hinzufügen von Tests zum Töten nicht wahrnehmen

Mutanten in uninteressanten Code als Verbesserung der insgesamt

Wirksamkeit der Testsuite, solche Mutanten neigen dazu, zu überleben und

als unproduktiv gekennzeichnet werden.

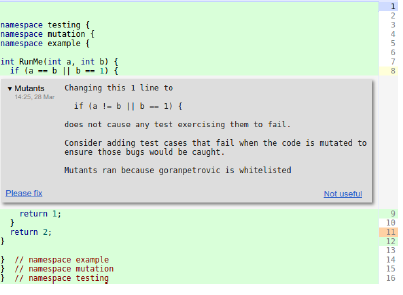


Abb. 3: Mutante im Code-Review-Tool gemeldet.

In diesem Abschnitt wird ein Ansatz zur Unterdrückung unproduktiver

Mutanten, basierend auf einer Reihe von Heuristiken zum Nachweis

aride (d.h. uninteressante) AST-Knoten. Es gibt einen Kompromiss

zwischen Richtigkeit und Nutzbarkeit der Ergebnisse; eine heuristische

kann eine Mutation in sehr wenigen nicht-ariden Knoten als verhindern

Nebeneffekt der Unterdrückung von Mutationen in vielen ariden Knoten.

Wir argumentieren, dass dies ein guter Kompromiss ist, weil die Zahl

der möglichen Mutanten ist um Größenordnungen größer als

was der Mutationsdienst vernünftigerweise dem

Entwickler innerhalb der bestehenden Entwickler-Tools. Darüber hinaus

Verhinderung nicht umsetzbarer Befunde ist wichtiger als

Berichterstattung über alle umsetzbaren Ergebnisse.

**3.1 Erkennen von ariden Knoten**

Um die Entstehung unproduktiver Mutanten zu verhindern,

der Mutationstestdienst identifiziert aride Knoten

im AST, die sich auf uninteressante Aussagen beziehen.

Beispiele für aride Knoten sind Aufrufe zur Speicherreservierung

Funktionen wie std:: vector:: reserve und schreiben in stdout;

diese werden in der Regel nicht durch Unit-Tests getestet.

Mutationsoperatoren erzeugen Mutanten basierend auf der AST von

ein Programm. Die AST enthält Knoten, die Anweisungen sind,

Ausdrücke oder Deklarationen und deren Kind-Eltern-Beziehungen

spiegeln ihre Verbindungen im Quellcode wider [24]. meisten

Compiler unterscheiden zwischen einfachen und zusammengesetzten AST

Knoten. Einfache Knoten haben keinen Körper; B. eine FunktionAlle

Ausdruck stellt einen Funktionsnamen und Argumente bereit,

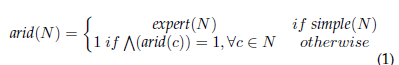
hat aber keinen Körper. Zusammengesetzte Knoten haben mindestens einen Körper;

Beispielsweise könnte eine for-Schleife einen Körper haben, während eine if

Anweisung könnte zwei haben - die dann und sonst Zweige.

Unser heuristikbasierter Ansatz zur Kennzeichnung von Knoten als arid

ist zweifach:



N 2 T ist hier ein Knoten im abstrakten Syntaxbaum T eines

Programm, einfach ist eine boolesche Funktion, die bestimmt, ob ein

Knoten ist ein einfacher oder zusammengesetzter Knoten (zusammengesetzte Knoten enthalten

ihre Kinder Knoten c), und Experte ist eine teilweise boolesche

Funktion, die eine Teilmenge einfacher Knoten in T auf die Eigenschaft, trocken zu sein, abbildet. Der erste Teil von Gleichung 1 funktioniert

auf einfachen Knoten, mit der Expertenfunktion, die kodiert

Wissen, das für jede Programmierung manuell kuratiert wird

Sprache und im Laufe der Zeit angepasst. Der zweite Teil arbeitet

auf zusammengesetzten Knoten und wird rekursiv definiert. Eine Verbindung

Knoten ist arid iff alle seine untergeordneten Knoten sind arid.

Die Expertenfunktion kennzeichnet einfache Knoten als arid und ist

basierend auf dem Feedback der Entwickler zu dem gemeldeten „Nicht nützlich“

Mutanten. Dies ist ein manueller Prozess: Wenn wir feststellen, dass ein

bestimmte Mutante ist in der Tat unproduktiv und dass eine ganze

Klasse solcher Mutanten sollte nicht erstellt werden, eine Regel wird hinzugefügt

zur Expertenfunktion. Dies ist eine Schlüsselkomponente der

Mutation Testing Service - ohne ihn würden Benutzer zu

frustriert über nicht umsetzbare Ergebnisse und Opt-out aus der

System insgesamt. Gezielte Mutation und sorgfältige Berichterstattung

von Mutanten waren entscheidend für die Annahme der Mutation

Testen bei Google. Bisher haben wir mehr als

einhundert Regeln für die Erkennung arider Knoten.

**3.2 Heuristische Expertenkategorien**

Die Expertenfunktion besteht aus verschiedenen Regeln, von denen einige

sind mutationsoperatorspezifisch und einige davon universell.

Wir unterscheiden zwischen Heuristiken, die die

Generierung von unkompilierbaren vs. kompilierbaren aber unproduktiven

Mutanten. Die meisten Heuristiken befassen sich mit der letzteren Kategorie, aber

Ersteres ist auch wichtig, vor allem in Go, wo die

Compiler ist sehr empfindlich gegenüber Mutationen (z.B.

import ist ein Compilerfehler). Für kompilierbare Mutanten haben wir

weitere Unterscheidung zwischen Heuristiken für äquivalente Mutanten,

tötbare Mutanten und redundante Mutanten, wie berichtet

in Tabelle 2.

Jede der vier heuristischen Kategorien enthält eine oder

eindeutigere Gruppen von Regeln, die wiederum eine enthalten

oder mehrere verwandte Regeln. Beispielsweise alle Regeln, die unterdrücken

Mutanten in Logging-Anweisungen (mehrere Regeln für mehrere

B. Arten von Protokollierungsanweisungen und -funktionen) bilden eine

Gruppe, weil sie alle für die Protokollierung gelten, und die gesamte

Gruppe zielt darauf ab, unproduktive tötbare Mutanten zu verhindern.

Häufigkeit gibt an, wie oft eine Kategorie auf eine

gegebene Änderungsliste. Eine detaillierte Liste der Regeln finden Sie unter

die ergänzenden Materialien, die online unter

< Produktionsmitarbeiter fügen Link ein >.

3.2.1 Heuristik zur Verhinderung nicht kompilierbarer Mutanten

Eine Mutante sollte ein syntaktisch gültiges Programm sein -

andernfalls würde es vom Compiler erkannt und würde

keinen Wert für das Testen hinzufügen. Es gibt bestimmte Mutationen,

besonders diejenigen, die Code löschen, die diese Gültigkeit verletzen

Prinzip. Ein Paradebeispiel ist das Löschen von Code in Go; irgendwelcher

unbenutzte Variable oder importiertes Modul erzeugt einen Compiler

Fehler. Die vorgeschlagene Heuristik sammelt alle verwendeten Symbole und

legt sie in einem Container, anstatt sie zu löschen, so dass sie

bleiben referenziert und der Compiler ist beschwichtigt.

3.2.2 Heuristik zur Verhinderung äquivalenter Mutanten

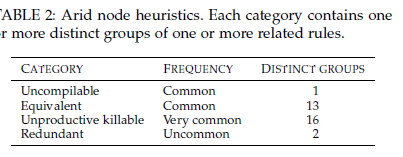
Äquivalente Mutanten, die semantisch äquivalent sind zu

das mutierte Programm, sind eine Plage im Mutationstest

und kann im Allgemeinen nicht automatisch erkannt werden. Doch

Es gibt einige Gruppen von äquivalenten Mutanten, die sein können

genau erkannt. In Java beispielsweise wird die Spezifikation



für die size-Methode einer java.util.Collection ist, dass es

gibt einen nicht negativen Wert zurück. Das bedeutet, dass Mutationen

wie collection.size () = = 0 7! Sammlung.size () < = 0

garantiert eine gleichwertige Mutante.

Ein weiteres Beispiel für diese Kategorie bezieht sich auf die Memoisierung.

Memoisierung wird oft verwendet, um die Ausführung zu beschleunigen, aber

seine Entfernung führt unweigerlich zur Erzeugung von äquivalenten

Mutanten. Die folgende Heuristik wird verwendet, um Memoisierung zu erkennen:

eine if-Anweisung ist eine Cache-Suche, wenn sie von der Form ist, wenn

a, ok: = x [v]; ok gibt a zurück, d.h. wenn ein Lookup in der Map

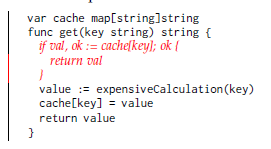
findet ein Element, der if-Block gibt dieses Element zurück (unter

B. Fehler in Go). Eine solche if-Anweisung ist eine

Cache-Lookup-Anweisung und wird vom Experten als trocken betrachtet

Funktion, ebenso wie sein ganzer Körper. Das folgende Beispiel zeigt eine

Cache-Suche in Go:



Das Entfernen der if-Anweisung entfernt nur das Zwischenspeichern, tut es aber

das Funktionsverhalten nicht verändern und somit ein äquivalentes

Mutante. Das Programm erzeugt immer noch die gleiche Ausgabe

für die gleiche Eingabe - wenn auch langsamer. Funktionstests sind nicht

erwartet, solche Änderungen zu erkennen.

Als drittes Beispiel vermeidet eine Heuristik in dieser Kategorie

Mutationen von Zeitangaben, da Unit-Tests selten testen

für Zeit, und wenn sie es tun, neigen sie dazu, gefälschte Uhren zu verwenden. Behauptungen

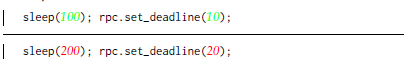
Aufrufen schlafähnlicher Funktionen, Festlegen von Fristen oder

Warten, bis Dienste bereit sind (wie gRPC [25] Server

Wait-Funktion, die immer in RPC-Servern aufgerufen wird, die

sind reichlich vorhanden in Googles Codebasis) gelten als arid von

die Expertenfunktion.



3.2.3 Heuristik zur Verhinderung unproduktiver tötbarer Mutanten

Nicht jeder Code ist gleich wichtig: Einige Codes können zu

tötbare Mutanten, aber die Tests, die sie töten, sind nicht wertvoll

und würde nicht von erfahrenen Entwicklern geschrieben werden; solcher

Mutanten sind schlechte Testziele. Beispiele für diese Kategorie sind

Inkremente von Werten in Überwachungssystem-Frameworks, niedrig

Level-APIs oder Flag-Änderungen: Diese sind leicht zu mutieren, einfach zu

Test für, und doch meist unerwünschte Testziele.

Eine gängige Methode zur Implementierung von Heuristik in dieser Kategorie

mit Funktionsnamen übereinstimmen soll; tatsächlich unterdrücken wir Mutanten in

ruft Hunderte von Funktionen auf, die für den größten Teil der Unterdrückungen durch die Expertenfunktion verantwortlich sind.

Das Paradebeispiel für diese Kategorie ist eine Heuristik, die

beliebiger Funktionsaufruf arid, wenn der Funktionsname mit dem

Präfixprotokoll oder das Objekt, für das die Funktion aufgerufen wird, ist

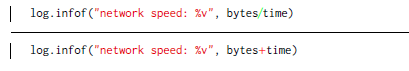
Logger genannt. Wir haben diese Heuristik durch zufällige Stichproben validiert

100 Knoten, die durch die Log-Heuristik arid markiert wurden,

und festgestellt, dass 99 tatsächlich richtig markiert waren, während man

hatte marginalen Nutzen. Insgesamt haben wir Fuzzy angesammelt

Namensunterdrückungsregeln für mehr als 200 Funktionsfamilien.



3.2.4 Heuristik zur Vermeidung redundanter Mutanten

Daran erinnern, dass die Mutation Testing Service erzeugt höchstens

eine Mutante pro Zeile und meldet eine eingeschränkte Teilmenge von

überlebende Mutanten während der Code-Überprüfung. Heuristik in diesem

Kategorie unterdrücken einige Mutanten, die redundant sind (dh,

funktionell äquivalent zu anderen Mutanten) aus zwei Gründen.

Erstens, während redundante Mutanten funktional äquivalent sind

zueinander, einige von ihnen sind leichter zu argumentieren über

als andere, so dass sie produktiver. Zweitens,

wenn ein Entwickler seine Änderungsliste aktualisiert und möglicherweise schreibt

Tests, um Mutanten zu töten, diese Änderung erstellt einen neuen Schnappschuss

und löst eine Wiederholung des Mutationsdienstes aus, wodurch das Testen

die Änderung und möglicherweise die Meldung neuer Mutanten. Reihenfolge

Entwicklerproduktivität und Benutzererfahrung zu verbessern,

Mutation Testing Service sollte konsequent generieren die

dieselbe Mutante aus einem Pool von gleich produktiven und

Vermeidung von Abweichungen von zuvor gemeldeten Mutanten, insbesondere

für unveränderte Zeilen zwischen Snapshots. Solche Divergenz

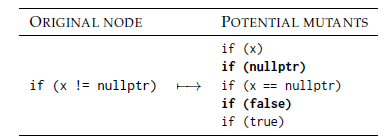
würde Verwirrung stiften, kognitiven Overhead einführen,

und damit geringere Entwicklerproduktivität.

Beispielsweise hat der LCR-Mutationsoperator in C++ eine

Sonderfall beim Umgang mit NULL (d.h. nullptr), weil

seiner logischen Äquivalenz mit false:



Die fett markierten Mutanten sind redundant, da die

Wert von nullptr entspricht false. Ebenso das Gegenteil

Beispiel, wo die Bedingung ist, wenn (nullptr = = x),

liefert redundante Mutanten für die linke Seite.

3.2.5 Erfahrungen mit Heuristik

In unserer Erfahrung der Anwendung von Heuristik, die höchste Produktivität

Gewinne resultierten aus drei implementierten Heuristiken

in den frühen Tagen: Unterdrückung von Mutationen bei der Protokollierung

Aussagen, zeitbezogene Vorgänge (z.B. Fristsetzung,

Timeouts, exponentielle Backoff-Spezifikationen usw.) und schließlich

Konfigurations-Flags. Der Großteil des frühen Feedbacks betraf

unproduktive Mutanten in einem solchen Code, der in

die Codebasis. Während es schwierig ist, genau zu messen, gibt es

ist ein starker Hinweis darauf, dass diese Unterdrückungen Konto für

Produktivitätssteigerungen von etwa 15% auf 80%. Zusätzlich

Heuristik und Verfeinerungen progressivley verbessert

Produktivität auf 89%.

Heuristik wird durch übereinstimmende AST-Knoten implementiert

mit den vollständigen Compilerinformationen, die für die Mutation verfügbar sind

Betreiber. Einige Heuristiken sind unsolide: sie beschäftigen

unscharfe Namensübereinstimmung und AST-Formen erkennen, aber kann

produktive Mutanten unterdrücken. Auf der anderen Seite, einige

Heuristik nutzen die vollständigen Typinformationen (wie Matching

java.util.HashMap:: size calls) und sind Sound. Ton

Heuristiken sind nachweislich korrekt, aber wir hatten viel

wichtigere Verbesserungen der wahrgenommenen Mutantennutzbarkeit

von unsoliden Heuristiken.

**4 MUTATIONSOPERATORAUSWAHLSTRATEGIEN**

Nach der Markierung arider Knoten im AST, der Mutationstest

Service generiert Mutanten für die verbleibenden, nicht ariden

Knoten. Dies beinhaltet zwei Herausforderungen. Zuerst nur generiert

Mutanten, die die Tests überleben, werden den Entwicklern gemeldet

während der Code-Überprüfung; Mutanten, die nicht überleben, nur verwenden

Rechenressourcen. Da viele Mutanten nicht

überleben die Tests und Mutagenese erzeugt nur eine einzige

Mutante pro Linie ist das Ziel, Mutanten zu schaffen, die eine

hohe Überlebenschancen. Ein iterativer Ansatz, bei dem nach

die erste Testrunde weitere Runden der Mutagenese könnte

für Zeilen ausgeführt werden, in denen Mutanten getötet wurden, würde die

Systeme ineffizient zu bauen und zu testen, und würde viel

länger wegen mehrerer Runden. In ähnlicher Weise werden alle

Mutanten pro Zeile sind rechnerisch zu teuer. Zweitens,

nicht alle überlebenden Mutanten sind gleich produktiv: abhängig

auf den Kontext, können bestimmte Mutation Betreiber produzieren

bessere Mutanten als andere. Daher ist das Ziel,

überlebende Mutanten, die eine hohe Chance haben, produktiv zu sein.

Eine effektive Mutationsoperatorauswahlstrategie nicht

stellt nur einen guten Kompromiss zwischen Produktivität und

Kosten, ist aber auch entscheidend für die Erstellung von Mutationsanalyseergebnissen

während der Codeprüfung umsetzbar.

Dieser Abschnitt stellt eine grundlegende Zufallsauswahlstrategie vor

das eine Mutante pro abgedeckter Linie erzeugt, unter Berücksichtigung

Informationen über aride Knoten und eine gezielte Auswahlstrategie,

die zusätzlich die bisherige Performance von

Mutationsoperatoren in ähnlichem Kontext (Abbildung 4).

**4.1 Zufallsauswahl**

Ein grundlegender, auf zufälligen Linien basierender Mutanten-Selektionsansatz könnte,

für jede Zeile in einer Änderungsliste eine der Mutanten auswählen

die für diese Linie gleichmäßig zufällig erzeugt werden können.

Alternativ könnte ein solcher Ansatz zufällig eine

Mutationspunkt in dieser Linie zuerst und dann zufällig auswählen

einen entsprechenden Mutationsoperator.

Daran erinnern, dass unser Ansatz für Mutationstests basiert

auf die Identifizierung von ariden Knoten, die nicht sein sollte

überhaupt mutiert. Darüber hinaus generiert unser Ansatz höchstens

eine einzelne Mutante pro Linie; keine zusätzlichen Mutanten erzeugt werden.

Listing 1 beschreibt unseren Zufallsauswahlalgorithmus

das diese beiden Designentscheidungen berücksichtigt. Die Mutation

Operatoren, die für eine bestimmte Sprache verfügbar sind, werden zufällig gemischt

und versuchte eins nach dem anderen, für jede abgedeckte, geänderte Zeile

entsprechend nicht-ariden Knoten in der Änderungsliste, bis ein

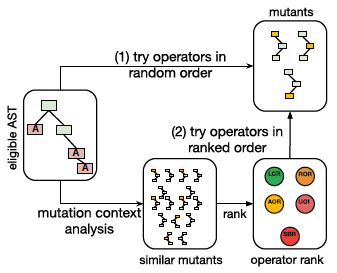


Abb. 4: Zufällige (1) vs. gezielte (2) Mutationsauswahl.

Mutante für diese Zeile erzeugt wird oder alle Operatoren wurden

versucht. Wenn mehrere Mutanten in einer Zeile erzeugt werden können, nur

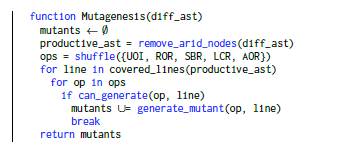
eine Mutante generiert wird, welche aber von der

zufälliges Shuffle und das AST selbst. Zum Beispiel die ROR

Mutationsoperator kann keine Mutante in einer Linie erzeugen, die

hat keine relationalen Operatoren, aber der SBR-Operator könnte -

die meisten Zeilen können gelöscht werden.



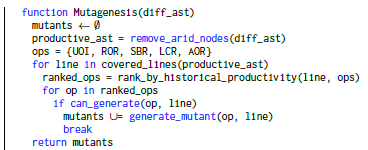
Listing 1: Zufallsauswahl mit Unterdrückung.

**4.2 Gezielte Auswahl**

Im Gegensatz zur Zufallsauswahl wird die gezielte Auswahl

Strategie ordnet die Mutationsoperatoren nach ihrer historischen Produktivität

unter Berücksichtigung des AST-Kontexts, wie in Auflistung 2 dargestellt.



Listing 2: Gezielte Auswahl mit Unterdrückung.

Das Ranking des Mutationsoperators für einen bestimmten AST-Knoten ist

basierend auf historischen Informationen, insbesondere Überlebensfähigkeit

und Produktivität. Die Überlebensfähigkeit eines Mutationsoperators ist die

Verhältnis der überlebenden Mutanten, die von diesem Operator in

einen gegebenen Kontext. Die Produktivität eines Mutationsoperators ist die

Verhältnis der von diesem Operator erzeugten produktiven Mutanten in einem

gegebenen Kontext. Die Produktivität basiert auf dem Feedback der Entwickler:

Während der Codeüberprüfung können Autoren und Prüfer Mutanten markieren

in einer Änderungsliste als produktiv oder unproduktiv dargestellt. Als

diese Entwickler verstehen den Kontext der Mutanten sie

kennzeichnen, im Gegensatz zu Teilnehmern, die eine Beschriftungsaufgabe ausführen

in einer Studie betrachten wir diese Information als starkes Signal.

Für jede Mutante der AST-Kontext, der die

Umgebung des AST-Knotens, der mutiert wurde, gespeichert wird

zusammen mit der Produktivität Feedback und ob die Mutante

getötet wurde oder nicht. Die gezielte Auswahlstrategie nutzt

diese Informationen, um AST-Knoten zu identifizieren, die ähnlich sind wie

die mutierte, basierend auf dem AST-Kontext. Die historische

Informationen über die für diese ähnlichen AST generierten Mutanten

Knoten wird dann verwendet, um die Mutation Operatoren ordnen, sondern

als eine zufällige Reihenfolge zu verwenden. Mutagenese wird dann versucht

in der resultierenden Reihenfolge, um die Wahrscheinlichkeit zu maximieren, dass die

Mutante wird überleben und produktiv sein.

**4.3 Mutationskontext**

Um historische Informationen über die Überlebensfähigkeit anzuwenden

und Produktivität, müssen wir entscheiden, wie ähnlich Kandidat

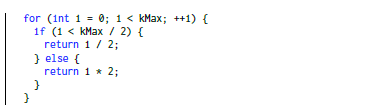
Mutationen werden mit vergangenen Mutationen verglichen. Wir definieren eine

Mutation ähnlich zu sein, wenn es in einem ähnlichen Kontext geschah,

z.B. Ersetzen eines relationalen Operators innerhalb einer if-Bedingung

das ist die erste Anweisung im Hauptteil einer for-Schleife, wie gezeigt

in Auflistung.



Listing 3: C++ -Schnipsel: eine if-Anweisung innerhalb einer for-Schleife.

Um die Ähnlichkeit des Kontexts von

zwei Mutationen, wir verwenden das Hashing-Framework für treestructured

von Tatikonda et al. [26], die

bildet einen ungeordneten Baum in eine Vielzahl einfacher Strukturen ab

als Pivots bezeichnet. Jedes Pivot erfasst Informationen

über die Beziehung zwischen den Knoten des Baums (siehe

Abschnitt 4.4).

Das Auffinden ähnlicher Mutationskontexte reduziert sich dann auf

Auffinden ähnlicher Pivot-Multisets. Identifizierung eines ähnlichen Drehpunkts

Multisets produzieren wir einen von MinHash [27] inspirierten Fingerabdruck

des Pivot-Multisets. Denn die Entfernung im Fingerabdruck

Raum korreliert mit der Entfernung im Baum Raum, können wir

ähnliche Mutationskontexte effizient finden, indem ähnliche

Fingerabdrücke von Knoten unter Mutation.

**4.4 Pivots aus ASTs generieren**

Um die komplizierte Beziehung zwischen Knoten zu erfassen

im AST übersetzen wir das AST in eine Vielzahl von Drehpunkten. A

Pivot ist ein Triplett von Knoten aus dem AST, das deren

Beziehung; für die Knoten u und v ist ein Pivot p Tupel (lca; u; v)

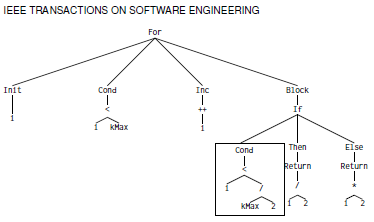
wobei lca der kleinste gemeinsame Vorfahr der Knoten u und

Abb. 5: AST für das Beispiel C++ in Listing 3.

*v.* Der Drehpunkt stellt einen Teilbaum des AST dar. Der Satz von

alle Pivots, die einen bestimmten Knoten betreffen, beschreiben den Baum

aus der Sicht dieses Knotens. Bei Mutationstests,

Wir sind nur an Knoten interessiert, die sich in der Nähe des Knotens befinden.

mutiert, so dass wir den Satz von Pivots auf Pivots beschränken

Knoten enthalten, die einen bestimmten Abstand vom Knoten haben

für Mutation in Betracht gezogen.

Im Beispiel zum Ersetzen eines relationalen Operators in einer

wenn Bedingung innerhalb eines Körpers der für Schleife in Auflistung 3, ein

pivot kann (if, Cond, \_) und ein anderer (Cond, i, kMax) sein. Alle

Kombinationen von zwei Knoten innerhalb einiger Entfernung von der

Knoten mutiert in der AST in Abbildung 5 und ihre niedrigste

gemeinsame Vorfahren machen Pivot-Strukturen.

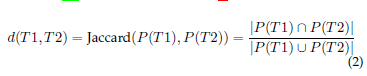
Pivot Multisets P bewahren präzise die strukturelle Beziehung

der Baumknoten (Eltern-Kind- und Vorfahren-Beziehungen),

also die Baumähnlichkeit zweier AST-Subbäume T1 und

T2 kann als Jaccard-Index des Drehpunkts gemessen werden

Multisets [26] wie in Gleichung 2 gezeigt.



**4.5 Fingerprinting Pivot Multisets**

Pivot-Multisets sind potenziell quadratisch in der Baumgröße, führend

zu kostspieligen Vereinigungs- und Kreuzungsoperationen. Sogar eine triviale

wenn Anweisung mit einer einzelnen Rückgabeanweisung große

Pivot-Sets und Set-Operationen werden unerschwinglich. Zu erleichtern

dass eine Fingerabdruckfunktion angewendet wird, um große

Multisets in Fingerabdrücke fester Größe schwenken.

Wir hash die Pivot-Sets auf einzelne Objekte, die die Form

Multiset von Vertretern für die Eingabe AST. Die Größe der

multiset kann groß sein, insbesondere für große Programme. Reihenfolge

um die Effizienz der weiteren Manipulation zu verbessern, verwenden wir eine

Signaturfunktion, die große Pivot-Hash-Sets in

kürzere Unterschriften. Die Signaturen werden später zur Berechnung verwendet

die Ähnlichkeit zwischen den Bäumen unter Berücksichtigung

nur der AST-Knotentyp und ignorieren alles andere, wie

Typdaten oder Namen der Bezeichner.

Wir verwenden eine einfache Hash-Funktion, um ein einzelnes Pivot zu hash *p =*

*(lca; u; v)* in einen von Tatikonda vorgeschlagenen festen Größenwert und

Parthasarathie [26].



Für a; b; c wir wählen kleine Primzahlen, und für K eine große Primzahl

das in 32 Bits passt. Um AST-Knoten hash zu können, weisen wir

spärliche Integer-Hash-Werte für verschiedene AST-Knotentypen in

jeder Sprache, z.B. einem C++ FunctionDecl ist 8500 zugeordnet,

und CXXMethodDecl 8600. Für Knoten im Pivot (lca; u; v)

verwenden wir diese zugewiesenen Hashes.

Beispiel: a = 17, b = 59, c = 83 und K =

15485863, können wir den Hash des Pivot berechnen (wenn, <, \_),

so einfach wie



wobei 32800 und 22400 die Integer-Hashwerte sind

IfStmt und BinaryOperator C++ AST Knoten zugewiesen.

Die Signatur für eine solche Tasche von Vertretern wird generiert

MinHashing-Technik verwenden. Der Satz von Pivots ist

permutiert und gehasht unter dieser Permutation. Zu minimieren

die falsch positiven und negativen (d.h. verschiedene Bäume Hash

zu ähnlichen Hashes oder umgekehrt), wird dies k-mal wiederholt,

was zu k-MinHashes führt.

Das Ziel ist, dass die Unterschriften für ähnliche

(multi) Sets und unähnlich für unähnliche. Jaccard-Ähnlichkeit

zwischen zwei Sätzen kann durch Vergleich geschätzt werden

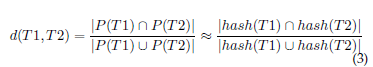
ihre MinHash-Signaturen auf die gleiche Weise [27], wie gezeigt

in Gleichung 3. Das MinHash-Schema kann als

Instanz von Locality-Sensitive-Hashing, wobei ASTs, die

einen geringen Abstand zueinander haben, werden in

Hashes, die diese Eigenschaft erhalten.



Wenn wir einen Knoten mutieren, berechnen wir seinen Pivot-Satz und

Hash es. Wir finden ähnliche AST-Kontexte mit dem nächsten Nachbarn

Suchalgorithmen. Wir beobachten, wie sich verschiedene Mutanten verhalten

in diesem Zusammenhang und welche Mutation Betreiber produzieren

die produktivsten und überlebendsten Mutanten. Das ist die Basis

zur gezielten Mutationsselektion.

**5 BEWERTUNG**

Um den Entwicklern einen Mehrwert zu bieten, bietet das Mutation Testing

Service bei Google muss wenige produktive Mutanten melden,

ausgewählt aus einem großen Pool von Mutanten - von denen die meisten

unproduktiv sind. Daran erinnern, dass eine produktive Mutante löst

ein wirksamer Test oder eine sonstige Verbesserung der Codequalität [11].

Daher ist unser Ziel zweifach. Zunächst wollen wir Mutanten auswählen

mit einer hohen Überlebensrate und Produktivität zur Maximierung

ihren Nutzen als Testziele. Zweitens wollen wir berichten

sehr wenige Mutanten, um den Rechenaufwand zu reduzieren und zu vermeiden

überwältigende Entwickler mit zu vielen Erkenntnissen.

Seit der Anwendung von Mutationstests auf die gesamte Codebasis

ist einfach nicht machbar, konzentrieren wir uns auf diff-basierte Mutation in unserer

Bewertung. Zusätzlich zur grundlegenden Designentscheidung der Anwendung

Mutationstests auf der Ebene der Changelists, zwei technische

Lösungen reduzieren die Anzahl der Mutanten: (1) Mutante

Suppression mit ariden Knoten und (2) one-per-line Mutante

Auswahl. Unsere Auswertung verwendet zwei Datensätze (Abschnitt 5.1) und

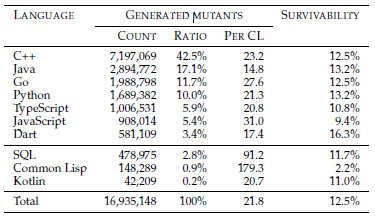
beantwortet vier Forschungsfragen. Die erste Forschungsfrage

betrifft die Wirksamkeit unserer beiden technischen Lösungen:

TABELLE 3: Zusammenfassung des mutierten Datensatzes. (Beachten Sie, dass SQL,

Common Lisp und Kotlin sind von unseren Analysen ausgeschlossen

wegen unzureichender Daten.)



\_ **RQ1 Mutantenunterdrückung**. Wie wirksam ist Mutante

Suppression mit ariden Knoten und 1-per-line Mutante

Auswahl? (Abschnitt 5.2)

Um den Einfluss der Mutationsbetreiberauswahl zu verstehen

auf Mutanten Überlebensfähigkeit und Produktivität in den verbleibenden

nicht-aride Knoten, betrachten wir historische Daten, einschließlich

Entwicklerfeedback. Wir wollen die folgenden zwei Antworten

Forschungsfragen:

\_ **RQ2 Mutante Überlebensfähigkeit**. Hat Mutationsoperator

Selektion beeinflussen die Wahrscheinlichkeit, dass eine generierte Mutante

überlebt die Testsuite? (Abschnitt 5.3)

\_ **RQ3 Mutantenproduktivität**. Hat Mutationsoperator

Auswahl Einfluss Entwicklerfeedback auf eine generierte

Mutant? (Abschnitt 5.4)

Nach Feststellung des Einflusses einzelner Mutationsoperatoren

auf Überlebensfähigkeit und Produktivität, die letzte Frage

ist, ob Mutationskontext verwendet werden kann, um beide zu verbessern.

Daher lautet unsere letzte Forschungsfrage wie folgt:

\_ **RQ4 Mutationskontext**. Führt kontextbasierte Auswahl von

Mutationsoperatoren verbessern die Überlebensfähigkeit von Mutanten und

Produktivität? (Abschnitt 5.5)

**5.1 Versuchsaufbau**

Für unsere Analysen haben wir zwei Datensätze erstellt, einen mit

Daten zu allen Mutanten und eine mit zusätzlichen Daten zu

Mutationskontext für eine Teilmenge aller Mutanten.

**Mutierter Datensatz.** Der mutierte Datensatz enthält 16.935.148

Mutanten in 10 Programmiersprachen: C++, Java, Go,

Python, TypeScript, JavaScript, Dart, SQL, Common Lisp,

und Kotlin. Tabelle 3 fasst den mutierten Datensatz zusammen und

gibt die Anzahl und das Verhältnis der Mutanten pro Programmierung an

Sprache, die durchschnittliche Anzahl der Mutanten pro Änderungsliste

und den Prozentsatz der Mutanten, die die Testsuite überleben.

Tabelle 4 schlüsselt die Zahlen nach Mutationsoperatoren auf.

Wir haben diesen Datensatz erstellt, indem wir Daten zu allen Mutanten gesammelt haben

die der Mutationstestdienst seit seiner

Einweihung, die sich auf das Datum bezieht, an dem wir die

Service breit verfügbar, nach der ersten Entwicklung der

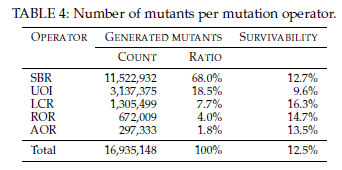
Dienst und seine Unterdrückungsregeln (siehe Abschnitt 3.2.5). Wir haben

keine Datenfilterung durchführen, daher liefert der Datensatz

Informationen über alle durchgeführten Mutationsanalysen.

Insgesamt betrachtete unsere Datenerhebung 776.740 Änderungslisten

die Teil des Code-Review-Prozesses waren. Für diese,



16.935.148 Mutanten wurden generiert, davon 2.110.489

gemeldet wurden. Von allen gemeldeten Mutanten erhielten 66.798

explizites Entwickler-Feedback. Für jede betrachtete Änderungsliste,

Der mutierte Datensatz enthält Informationen über:

\_ betroffene Dateien und betroffene Zeilen,

\_ Testziele, die die betroffenen Leitungen testen,

\_ für jede der betroffenen Linien erzeugte Mutanten,

\_ Testergebnisse für die Datei in der mutierten Zeile und

\_ Mutationsoperator und Kontext für jede Mutante.

Unsere Analyse zielt darauf ab, die Wirksamkeit und wahrgenommene

Produktivität von Mutanten und Mutationsoperatoren über die Programmierung hinweg

Sprachen. Beachten Sie, dass unser mutierter Datensatz wahrscheinlich ist

spezifisch für den Codestil und die Überprüfungspraktiken von Google. Doch

der Code-Stil ist weit verbreitet [28], und die moderne

Der Code-Review-Prozess wird in der gesamten Branche eingesetzt [29].

Informationen zur Überlebensfähigkeit von Mutanten pro Programmierung

Sprache oder Mutationsoperator kann direkt extrahiert werden

aus dem Datensatz und ermöglicht es uns, Forschung zu beantworten

Fragen **RQ1**, **RQ2** und **RQ3**.

**Kontextdatensatz.** Der Kontextdatensatz enthält 4.068.241

Mutanten (eine Teilmenge des mutierten Datensatzes) für die Top-4

Programmiersprachen: C++, Java, Go und Python. Jeder

Mutante in diesem Datensatz ist angereichert mit den Informationen von

ob unsere kontextbasierte Auswahlstrategie

diese Mutante ausgewählt. Bei der Erzeugung von Mutanten würden wir

auch die kontextbasierte Vorhersage ausführen, und wir beharrten

die Vorhersageinformation zusammen mit den Mutanten. Wenn die

zufällig ausgewählten Operator war in der Tat, was die Vorhersage

Service ausgewählt, ist diese Mutante derjenige mit der höchsten

vorhergesagter Wert. Für jede Mutante enthält der Datensatz:

\_ alle Informationen aus dem mutierten Datensatz,

\_ vorhergesagte Überlebensfähigkeit und Produktivität für jede Mutation

in ähnlichem Zusammenhang und

\_ Information, ob die Mutante die höchste

vorhergesagte Überlebensfähigkeit/Produktivität.

Wir haben diesen Datensatz mithilfe unserer kontextbasierten Mutation erstellt

Selektionsstrategie während der Mutagenese an allen Mutanten

während eines begrenzten Zeitraums. Während dieser Zeit haben wir

die Mutanten automatisch mit Anmerkungen versehen, die angeben, ob

eine Mutante würde durch die kontextbasierte Mutation ausgewählt

Selektionsstrategie zusammen mit dem mutierten Ergebnis in Bezug auf

der Überlebensfähigkeit und Produktivität. Dieser Datensatz ermöglicht die

Bewertung unserer kontextbasierten Mutationsselektionsstrategie

und ermöglicht es uns, Forschungsfragen **RQ4** zu beantworten.

**Experimentelle Maßnahmen:** Das Überleben der ersten Testsuite ist ein

Voraussetzung für das Auftauchen einer Mutante, aber Überlebensfähigkeit allein

ist kein gutes Maß für die Produktivität von Mutanten. Entwickler

Rückmeldung, dass eine Mutante tatsächlich (un-) produktiv ist

ist ein stärkeres Signal.

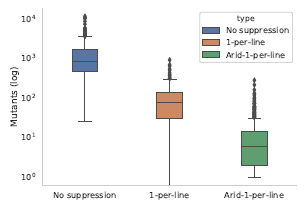


Abb. 6: Anzahl der generierten Mutanten pro Änderungsliste für nein

Suppression (traditionelle Mutagenese), 1-per-line und arid-1-

per-line (unser Ansatz). (Beachten Sie die logarithmisch skalierte vertikale Achse.)

Wir messen die Produktivität von Mutanten über das gesammelte Feedback der Benutzer

aus Kritik (Abschnitt 2.4), wo jeder gemeldete Mutant

zeigt ein Please fix (produktive Mutante) und ein Not

nützliche (unproduktive Mutante) Verbindung. Bitte korrigieren entspricht

eine Aufforderung an den Autor einer Änderungsliste, den Test zu verbessern

Suite basierend auf der gemeldeten Mutante; nicht sinnvoll entspricht

auf einen Fehlalarm oder generell eine nicht umsetzbare Codefindung.

82% aller gemeldeten Mutanten mit Feedback waren markiert

als produktiv von Entwicklern. Beachten Sie, dass dieses Verhältnis ein

über den gesamten Datensatz aggregieren. Seit der Einweihung

des Mutationstestdienstes hat die Produktivität zugenommen

im Zeitverlauf von 80% auf 89%, weil wir die

Feedback zu unproduktiven Mutanten und verursachte Unterdrückung

Regeln für die Expertenfunktion, beschrieben in Abschnitt 3. Das

bedeutet, dass spätere Mutationen von Knoten, in denen Mutanten waren

als unproduktiv empfunden wird unterdrückt, wodurch

weniger unproduktive Mutanten im Laufe der Zeit. Gemeldete Mutanten

ohne explizites Entwickler-Feedback nicht berücksichtigt

die Produktivitätsanalyse.

**5.2 RQ1 Mutantensuppression**

Um unseren Mutanten-Suppression-Ansatz zu vergleichen

mit der traditionellen Mutagenese, wir (1) stichprobenartig

5.000 Änderungslisten aus dem mutierten Datensatz, (2) ermittelt

wie viele Mutanten die traditionelle Mutagenese hervorbringt und

(3) das Ergebnis mit der Anzahl der erzeugten Mutanten verglichen

durch unseren Ansatz. (Da traditionelle Mutationsanalyse ist

unerschwinglich teuer im Maßstab, haben wir unser System angepasst

nur alle Mutanten für die ausgewählten Änderungslisten generieren.) Abbildung

6 zeigt die Ergebnisse für drei Strategien: keine Unterdrückung

(traditionell), wählen Sie eine Mutante pro Zeile und wählen Sie eine Mutante

pro Zeile nach Ausschluss arider Knoten (unser Ansatz). Wir

Einbeziehung des 1-pro-Linie-Ansatzes in die Analyse zur Bewertung

den individuellen Beitrag der Unterdrückung arider Knoten,

über die Probenahme einer Mutante pro Zeile hinaus.

Wie in Tabelle 5 gezeigt, ist die mediane Anzahl der generierten

Mutanten ist 820 für traditionelle Mutagenese, 77 für 1-pro-Linie

Auswahl, und nur 7 für arid-1-per-line Auswahl. Daher,

unser Mutanten-Suppression-Ansatz reduziert die Anzahl der

Mutanten um zwei Größenordnungen. Tabelle 5 zeigt auch die

Ergebnisse für einen Mann-Whitney U-Test, der bestätigt, dass die

Verteilungen sind statistisch signifikant unterschiedlich.

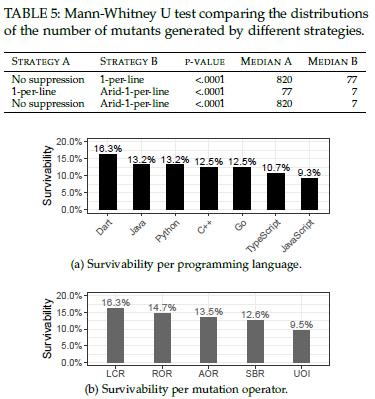


Abb. 7: Mutante Überlebensfähigkeit.

Unser Mutanten-Suppression-Ansatz erzeugt weniger als

20 Mutanten für die meisten Changelists; das 25. und 75. Perzentil

sind 3 bzw. 19. Im Gegensatz dazu ist der 25.

und 75. Perzentile für 1-per-line 31 und 138 Mutanten sind.

Traditionelle Mutagenese erzeugt mehr als 450 Mutanten

für die meisten Änderungslisten (das 25. und 75. Perzentil sind 460

bzw. 1734), was weiter unterstreicht, dass dieser Ansatz

ist unpraktisch, auch auf der Ebene der Änderungsliste. Präsentieren

Hunderte von Mutanten, von denen die meisten nicht umsetzbar sind, um

ein Entwickler würde fast sicher dazu führen, dass Entwickler

Mutationstests ganz einzustellen.

**RQ1:** Arid-Knoten-Unterdrückung und 1-pro-Zeile-Auswahl deutlich

reduzieren die Anzahl der Mutanten pro Änderungsliste, mit einem

Median von nur 7 Mutanten pro Changelist (im Vergleich zu 820

Mutanten für die traditionelle Mutagenese).

**5.3 RQ2 Mutantenüberlebensfähigkeit**

Mutante Überlebensfähigkeit ist wichtig, weil wir erzeugen bei

am meisten eine einzelne Mutante pro Linie - wenn diese Mutante getötet wird, nein

andere Mutante erzeugt wird. Um umsetzbar zu sein, müssen Mutanten

so bald wie möglich im Rahmen der Codeüberprüfung gemeldet werden,

wie in Abschnitt 4 beschrieben. Daher wollen wir die

Mutante Überlebensfähigkeit, weil es direkt die Zahl beeinflusst

der gemeldeten Mutanten.

Insgesamt werden 87,5% aller generierten Mutanten durch

die erste Testsuite. Beachten Sie, dass dies nicht dasselbe ist wie

der traditionelle Mutationsscore [30] (Verhältnis der getöteten Mutanten

zur Gesamtzahl der Mutanten), da Mutagenese

probabilistisch und erzeugt nur eine Teilmenge aller Mutanten. Das

bedeutet, dass nur ein Bruchteil aller möglichen Mutanten erzeugt wird

und ausgewertet, und viele andere Mutanten werden nie erzeugt

weil sie ariden Knoten zugeordnet sind.

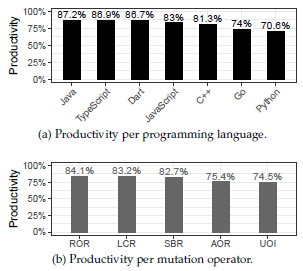


Abb. 8: Mutantenproduktivität.

Die Tabellen 3 und 4 zeigen die Verteilung der Anzahl der Mutanten

und mutante Überlebensfähigkeit, aufgeschlüsselt durch Programmierung

Sprache und Mutation Operator. Abbildung 7 visualisiert

die mutierten Überlebensfähigkeitsdaten. Weil die SBR-Mutation

-Operator kann auf fast jeden nicht-ariden Knoten im

Code, ist es keine Überraschung, dass diese Mutation Operator dominiert

die Anzahl der Mutanten, die etwa 68% aller

Mutanten. Während SBR ein produktiver und vielseitiger Mutationsoperator ist,

es ist auch die zweitniedrigste Wahrscheinlichkeit, die Testsuite zu überleben:

falls zutreffend für eine Änderungsliste, werden SBR-Mutanten gemeldet

bei Code-Review mit einer Wahrscheinlichkeit von 12,6%. Insgesamt

Mutante Überlebensfähigkeit ist ähnlich über Mutation Betreiber,

mit einer bemerkenswerten Ausnahme von UOI, die eine Überlebensfähigkeit hat

von nur 9,5%. Mutante Überlebensfähigkeit ist auch ähnlich über

Programmiersprachen mit Ausnahme von Dart, dessen

mutierte Überlebensfähigkeit ist deutlich höher. Wir vermuten, dass

Dies liegt daran, dass Dart hauptsächlich für die Webentwicklung verwendet wird

die ihre eigenen Testherausforderungen hat.

**RQ2:** Unterschiedliche Mutationsoperatoren führen zu unterschiedlichen Mutanten

Überlebensfähigkeit; zum Beispiel ist die Überlebensrate von LCR fast

doppelt so hoch wie die von UOI.

**5.4 RQ3 Mutantenproduktivität**

Mutantenproduktivität ist das wichtigste Maß, denn

es misst direkt den Nutzen einer gemeldeten Mutante. Seitdem

Wir erzeugen nur eine einzelne Mutante in einer Linie, diese Mutante

idealerweise sollte nicht nur die Test-Suite überleben, sondern auch

produktiv, sodass Entwickler die Testsuite verbessern können

oder der Quellcode selbst. Angesichts der hohen Genauigkeit von Google

und Anforderungen an die Umsetzbarkeit von Oberflächencodebefunden

bei Code-Reviews setzen wir auf Entwickler-Feedback als

beste verfügbare Messgröße für die Produktivität von Mutanten. Insbesondere

betrachten wir eine Mutante als einen Entwickler, der mit Please fix markiert ist

produktiver zu sein als andere. Ebenso betrachten wir eine

Mutante ein Entwickler markiert mit Nicht nützlich, um weniger produktiv zu sein

als andere. Wir vergleichen die Mutantenproduktivität

über Mutationsoperatoren und Programmiersprachen hinweg.

Abbildung 8 zeigt die Ergebnisse, was darauf hinweist, dass die Produktivität der Mutanten

ist ähnlich über Mutation Operatoren, mit AOR

und UOI-Mutanten, die merklich weniger produktiv sind. Dafür

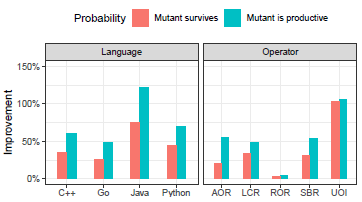


Abb. 9: Verbesserungen durch kontextbasierte Auswahl.

(0% Verbesserung entspricht zufälliger Auswahl.)

Beispiel: ROR-Mutanten sind in 84,1% der Fälle produktiv,

während, UOI-Mutanten sind nur produktiv 74,5% der

Zeit. Die Unterschiede zwischen Programmiersprachen sind

noch ausgeprägter, wobei Java-Mutanten produktiv sind

87,2% der Zeit, im Vergleich zu Python-Mutanten, die

produktiv 70,6% der Zeit. Dies könnte an Code liegen

Konventionen, gemeinsame Sprachverwendungsszenarien, Testen

Frameworks oder einfach das Fehlen von Heuristiken. Wir haben gefunden

dass Python-Code erfordert in der Regel mehr Tests wegen der

das Fehlen des Compilers. Im Gegensatz zu Python, die meist verwendet wird

für Backends sind JavaScript, TypeScript und Dart überwiegend

verwendet in Frontend-Code, der radikal anders ist.

**RQ3:** ROR-, LCR- und SBR-Mutanten zeigen eine ähnliche Produktivität;

während AOR und UOI Mutanten eine merklich geringere Produktivität aufweisen.

**5.5 RQ4 Mutationskontext**

Wir untersuchen, ob kontextbasierte Selektion der Mutation

Operatoren verbessert Mutanten Überlebensfähigkeit und Produktivität.

Konkret ermitteln wir, ob kontextbasierte Auswahl

von Mutationsoperatoren erhöht die Wahrscheinlichkeit eines generierten

Mutante, um zu überleben und zu einer Bitte fix Anfrage führen,

im Vergleich zur Zufallsauswahl-Baseline.

Abbildung 9 zeigt, dass die Auswahl von Mutationsoperatoren

zum AST-Kontext des Mutationsknotens im Wesentlichen

erhöht die Überlebenswahrscheinlichkeit der erzeugten Mutante

und zu einer Bitte korrigieren Sie die Anforderung. Während Verbesserungen

variieren über Programmiersprachen und Mutationen hinweg

Operatoren, die kontextbasierte Auswahl übertrifft konsequent

zufällige Auswahl. Die größten Produktivitätssteigerungen

werden für UOI, AOR und SBR erreicht, die

vor allem Mutanten. Intuitiv bedeuten diese Verbesserungen

dass die kontextbasierte Auswahl zu doppelt so vielen produktiven Ergebnissen führt

UOI-Mutanten (aus allen generierten Mutanten), wenn

im Vergleich zur Zufallsauswahl. Abbildung 9 zeigt auch, was

inwieweit diese Verbesserungen darauf zurückzuführen sind, dass

einfach mehr Mutanten überleben. Seit den Verbesserungen für

Produktivitätssteigerung noch stärker als die für Überlebensfähigkeit,

kontextbasierte Auswahl führt nicht nur zu mehr gemeldeten

Mutanten aber auch in höherer Produktivität dieser Mutanten.

Insgesamt steigt die Überlebensrate um über 40% und die

Wahrscheinlichkeit, dass ein Prüfer nach einer generierten Mutante fragt, um

fest erhöht werden um fast 50%.

Es ist wichtig, diese Verbesserungen in einen Kontext zu stellen.

Probabilistische diff-basierte Mutationsanalyse aggressiv trimmt

die Anzahl der in Betracht gezogenen Mutanten von Tausenden in

eine repräsentative Datei zu wenigen, und ermöglicht Mutanten zu sein

Entwickler effektiv als potenzielle Testziele präsentiert.

Der Zufallsauswahlansatz führt zu weniger Überlebenden

Mutanten geringerer Produktivität.

**RQ4:** Kontextbasierte Auswahl verbessert die Wahrscheinlichkeit, dass ein

generierte Mutante überlebt um mehr als 40% und die Wahrscheinlichkeit

dass eine erzeugte Mutante um fast 50% produktiv ist.

**6 DAMIT VERBUNDENE ARBEITEN**

Es gibt mehrere Forschungsvenen, die damit zusammenhängen

arbeiten. Nur et al. einen AST-basierten Programmkontext vorgeschlagen

Modell zur Vorhersage der Mutantenwirksamkeit [31]. Fernandez

et al. entwickelt verschiedene Regeln für Java-Programme zu erkennen

äquivalente und redundante Mutanten [32]. Die ersten Ergebnisse

sind vielversprechend für die Entwicklung von Auswahlstrategien, die besser abschneiden

zufällige Auswahl. Weiter, Zhang et al. Gebrauchtmaschine

Lernen, Mutationswerte vorherzusagen, beide auf aufeinanderfolgenden

Versionen eines bestimmten Projekts und projektübergreifend [33]. Schließlich

das PIT-Projekt macht Mutationstests durch Üben nutzbar

Entwickler und hat sich in der Branche durchgesetzt [16].

Es wurde viel Fokus auf Rechenkosten und

das äquivalente Mutantenproblem [34]. Es gibt viel Fokus

zur Vermeidung redundanter Mutanten, was zu einem Anstieg der

Rechenkosten und Inflation des Mutationscores [35],

und stattdessen schwer nachweisbare Mutanten [36], [37] oder

Dominatormutanten [38]. Mutanten-Subsumtionsdiagramme haben

ähnliche Ziele, aber mutierte Produktivität ist viel unschärfer

als Marktbeherrschung oder Subsumption.

Die Wirksamkeit für Mutanten wird in erster Linie in Bezug auf definiert

Redundanz und Äquivalenz. Dieser Ansatz berücksichtigt nicht

die Vorstellung, dass nicht-reduntante Mutanten unproduktiv sein könnten

oder dass äquivalente Mutanten produktiv sein können [39].

Aus unserer Erfahrung wurde die Meldung äquivalenter Mutanten

ein weitaus einfacheres Problem als die Meldung unproduktiver nichtreduntanter

und nicht-äquivalente Mutanten.

Unser Ansatz zur gezielten Mutantenselektion (Abschnitt 4)

vergleicht den Kontext von Mutanten mit Baum-Hashes.

Die spezifische Umsetzung wurde durch die Notwendigkeit der Konsistenz angetrieben

und Effizienz, um es möglich zu machen, zu suchen

bis ähnliche AST-Kontexte in Echtzeit während der Mutantenerstellung.

Insbesondere müssen die Hash-Abstände über

Zeit, die gezielte Auswahl zu verbessern. Es gibt Ansätze

zur Erkennung von Software-Klonen [40], die ebenfalls Treedistances verwenden

(z. B. [41], [42], [43], [44], [45]). Zur Alternative

Entfernungsmessungen können für die Anwendung skaliert werden bei

Google und ob sie die angestrebten

Auswahl bleibt in zukünftigen Arbeiten festgelegt werden.

Dieser Ansatz ähnelt baumbasierten Ansätzen in

Erkennung von Software-Klonen [40], mit der festgestellt werden soll, dass ein

Codefragment ist eine Kopie eines Originalcodes, mit oder

ohne Änderung. Die AST-basierten Techniken können erkennen,

zusätzliche Kategorien von Änderungen wie Bezeichnername

Änderungen oder Typ-Aliase, die Token-basierte Erkennung nicht,

und die Unempfindlichkeit gegenüber Variablennamen ist wichtig

für den Mutationskontext. Klonerkennung unterscheidet sich jedoch

drastisch in seinem Ziel: es kümmert sich um die Erkennung von Code mit dem

gleiche Semantik, trotz der syntaktischen Änderungen

dazu. Während die Klonerkennung möglicherweise erkennen möchte, dass ein

Algorithmus wurde kopiert und dann leicht verändert, z.B.

eine Rekursion, die in einen äquivalenten iterativen Algorithmus umgeschrieben wurde,

Mutationstest Kontext kümmert sich nur um die benachbarten

AST-Knoten: im iterativen Algorithmus der produktivste

Mutanten werden diejenigen sein, die zuvor in einem solchen Code gediehen sind, nicht

diejenigen, die für einen rekursiven Algorithmus gediehen. Reihenfolge

ähnliche AST-Kontexte in Echtzeit nachschlagen, als Mutanten

erstellt werden, benötigen wir eine schnelle Methode, die Hash bewahrt

Entfernung über die Zeit. Für diese Konsistenz und Effizienz

Gründen haben wir uns für den beschriebenen Tree-Hashing-Ansatz entschieden.

**7 SCHLUSSFOLGERUNGEN**

Mutationstests haben das Potenzial, Software effektiv zu leiten

Testen und Verbessern der Softwarequalität. Doch viele

Mutanten stellen unproduktive Testziele dar; Schreiben von Tests für

sie nicht Testsuite Wirksamkeit verbessern und, noch schlimmer,

wirkt sich negativ auf die Wartbarkeit von Tests aus.

In den letzten sechs Jahren haben wir eine skalierbare

Mutationstestansatz und Mutantenunterdrückungsregeln

dass das Verhältnis der produktiven Mutanten erhöht, wie beurteilt

von Entwicklern. In den frühen Phasen des Projekts, die erste

Mutantenunterdrückungsregeln verbesserten das Verhältnis der produktiven

Mutanten von 15% bis 80%. Da das Produkt reifte, zusätzliche

Mutantenunterdrückungsregeln verbesserten die Produktivität

auf 89%. Drei Strategien waren der Schlüssel zum Erfolg. Erstens, wir

entwickelt eine inkrementelle Mutation Teststrategie, Berichterstattung

höchstens eine Mutante pro Codezeile - Ziellinien, die

geändert und abgedeckt werden. Zweitens haben wir einen Satz erstellt

regelbasierte Heuristik zur Mutantenunterdrückung, basierend auf

Entwicklerfeedback und manuelle Analysen. Drittens haben wir

einen probabilistischen, gezielten Mutantenselektionsansatz, der

Mutationskontext und historische Mutationsergebnisse.

Angesichts des Erfolgs unseres Mutationstestansatzes und

das positive Entwickler-Feedback, erwarten wir, dass weitere

Übernahme durch Entwicklungsteams führt zu zusätzlichen

Verfeinerungen der Unterdrückungs- und Selektionsstrategien. Darüber hinaus

ein wichtiger Aspekt unserer laufenden Forschung ist

die langfristigen Auswirkungen von Mutationstests auf

Entwicklerverhalten [20].