Практическое тестирование мутаций в большом масштабе

**Аннотация** — Мутационный анализ оценивает адекватность набора тестов путем измерения его способности обнаруживать небольшие искусственные ошибки, систематически вносимые в тестируемую программу. Мутационный анализ считается одним из самых сильных критериев адекватности теста. Мутационное тестирование основано на мутационном анализе и представляет собой метод тестирования, в котором мутанты используются в качестве целей тестирования для создания или улучшения набора тестов. Мутационное тестирование долгое время считалось неразрешимой задачей, поскольку огромное количество мутантов, которые могут быть созданы, представляет собой непреодолимую проблему — как с точки зрения человеческих, так и вычислительных усилий. Это препятствовало принятию мутационного тестирования в качестве отраслевого стандарта. Например, кодовая база Google насчитывает два миллиарда строк кода, и ежедневно выполняется более 150 000 000 тестов. Традиционный подход к мутационному тестированию не подходит для такой среды; даже существующие решения для ускорения анализа мутаций недостаточны, чтобы сделать его вычислительно осуществимым в таком масштабе.

Для решения этих проблем в данной статье представлен масштабируемый подход к мутационному тестированию, основанный на следующих основных идеях:

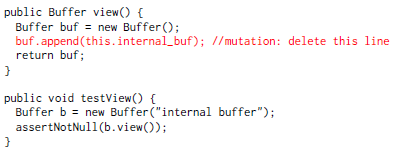
(1) тестирование мутаций проводится постепенно, при этом во время проверки кода мутируется только измененный код , а не вся база кода;

(2) мутанты фильтруются, удаляя мутанты, которые могут оказаться нерелевантными для разработчиков, и ограничивая количество мутантов в строке и в процессе проверки кода; (3) мутанты отбираются на основе исторических показателей операторов мутации, что дополнительно устраняет ненужные мутанты и улучшает качество мутантов. Эта статья эмпирически подтверждает предложенный подход, анализируя его эффективность в условиях проверки кода, используемой более чем 24 000 разработчиками в более чем 1000 проектах. Результаты показывают, что предложенный подход производит на порядки меньше мутантов и что контекстно-ориентированная фильтрация и отбор мутантов улучшают качество и эффективность мутантов. В целом, предлагаемый подход представляет собой структуру мутационного тестирования, которая легко интегрируется в рабочий процесс разработки программного обеспечения и применима к промышленным объектам любого размера.

**Индексные термины** — мутационное тестирование, покрытие кода, эффективность тестирования.

**1. ВВЕДЕНИЕ**

Тестирование программного обеспечения является преобладающим методом обеспечения качества программного обеспечения, и существуют различные подходы для оценки эффективности набора тестов (т. е. способности набора тестов обнаруживать дефекты программного обеспечения). Распространенным подходом является покрытие кода, которое широко используется в Google [1] и измеряет степень, в которой набор тестов проверяет программу. Покрытие кода интуитивно понятно, не требует больших затрат на вычисления и хорошо поддерживается инструментами коммерческого уровня. Однако само по себе покрытие кода недостаточно и может дать ложное ощущение эффективности, особенно если программные операторы покрыты, но их ожидаемый результат не утверждается в [2], [3]. Альтернативным подходом, устраняющим это ограничение, является анализ мутаций, который систематически вносит в программу искусственные ошибки, называемые мутантами, и измеряет способность набора тестов их обнаруживать [4]. Адреса анализа мутаций широко считаются лучшим подходом для оценки эффективности набора тестов [5], [6], [7]. Мутационное тестирование — это итеративный подход к тестированию, который основан на анализе мутаций и использует необнаруженные мутанты в качестве конкретных целей тестирования для управления процессом тестирования. В качестве конкретного примера рассмотрим следующее полностью покрытое, но слабо протестированное представление функции :



Тест выполняет функцию, но не проверяет ее влияние на возвращаемый буфер. В этом случае анализ мутаций превосходит покрытие кода: даже несмотря на то, что строка, добавляющая некоторый контент в buf , покрыта, разработчик не информируется о том, что ни один тест не проверяет ее влияние. Мутация удаления операторов, выделенная в примере кода, явно указывает на эту слабость тестирования: тест не завершается неудачей при вставке этого искусственного дефекта. Google всегда стремится улучшить качество тестирования и поэтому решил внедрить и развернуть мутационное тестирование для оценки его эффективности. Однако масштаб кодовой базы Google, насчитывающей примерно 2 миллиарда строк кода, сделал традиционный подход к мутационному тестированию невозможным: более 150 000 000 тестов в день контролируют 40 000 изменений в этой кодовой базе, гарантируя, что 14 000 непрерывных интеграций остаются работоспособными. ежедневно [8], [9]. Во-первых, систематическое изменение всей базы кода или даже отдельных проектов приводит к созданию значительного количества мутантов, каждый из которых потенциально требует выполнения множества тестов. Во-вторых, ни традиционно рассчитываемый коэффициент обнаружения мутантов, который количественно определяет эффективность набора тестов, ни простое представление разработчику всех мутантов, которые уклонились от обнаружения, не будут иметь практического значения. Учитывая, что разрешение одного мутанта занимает несколько минут [10], [11], необходимые усилия разработчика для разрешения всех необнаруженных мутантов будут непомерно дорогими, даже в небольших масштабах.

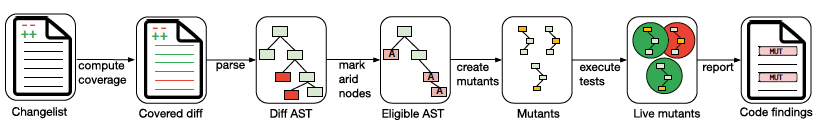


Рис. 1: Служба мутационного тестирования. Для данного списка изменений вычисляется покрытие линии и код анализируется в AST. Для узлов AST, охватывающих покрытые линии, засушливые узлы помечаются с использованием эвристики обнаружения засушливых узлов, и мутируются только незасушливые (подходящие) узлы. Сгенерированные мутанты тестируются, а о выживших мутантах сообщают как результаты кода.

Что еще хуже, даже при применении методов выборки для существенного сокращения количества мутантов разработчики Google изначально классифицировали 85% зарегистрированных мутантов как непродуктивные. Непродуктивный мутант либо тривиально эквивалентен исходной программе, либо его можно обнаружить, но добавление теста для него не улучшит набор тестов [11].

Например, изменение начальной емкости коллекции Java (например, new ArrayList(64) 7! new ArrayList(16)) создает непродуктивный мутант. Хотя можно написать тест, который проверяет емкость коллекции или ожидаемое выделение памяти, это непродуктивно. Фактически, вполне возможно, что эти тесты, если они будут написаны и добавлены, даже окажут негативное влияние, поскольку их природа обнаружения изменений (в частности, тестирование текущей реализации, а не спецификации) нарушает лучшие практики тестирования и приводит к нестабильности тестов и ложным тревогам. Столкнувшись с двумя основными проблемами при проведении мутационного тестирования — вычислительными затратами на мутационный анализ и тем фактом, что большинство мутантов непродуктивны — мы разработали подход к мутационному тестированию, который является масштабируемым и пригодным для использования, основанным на трёх основных идеях:

1) Наш подход выполняет мутационное тестирование изменений кода: он учитывает только измененные строки кода и сообщает о мутантах во время проверки кода (раздел 2, на основе нашей предыдущей работы [12]). Это значительно сокращает количество линий, в которых создаются мутанты, и соответствует единице работы разработчика, для которой желательны дополнительные тесты.

2) Наш подход использует подавление транзитивных мутантов: он использует эвристику, основанную на отзывах разработчиков (раздел 3, на основе нашей предыдущей работы [12]). Отзывы более 20 000 разработчиков о тысячах мутантов за шесть лет позволили нам разработать эвристику подавления мутантов, которая снижает долю непродуктивных мутантов с 85% до 11%.

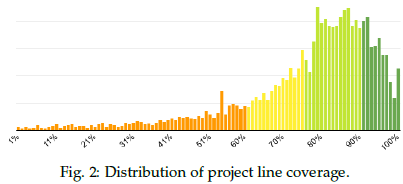
3) Наш подход использует вероятностный целевой отбор мутантов: он сообщает об ограниченном количестве мутантов на основе исторических характеристик мутантов, что позволяет избежать непродуктивных мутантов (раздел 4). Наша оценка предложенного подхода включала 760 000 изменений кода и 2 миллиона мутантов, о которых сообщалось во время проверки кода, из общего числа почти 17 миллионов сгенерированных мутантов (раздел 5). Результаты показывают, что наш подход делает тестирование мутаций возможным и действенным даже для сред разработки программного обеспечения промышленного масштаба.

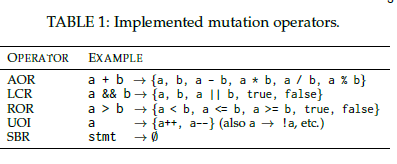
**2 ТЕСТИРОВАНИЕ МУТАЦИЙ В G OGLE**

Мутационное тестирование в Google сталкивается с проблемами масштаба, как с точки зрения времени вычислений, так и с точки зрения интеграции в рабочий процесс разработчика. Несмотря на то, что существующие работы по выборочной мутации и другим оптимизациям [13] могут существенно сократить количество мутантов, которые необходимо проанализировать, вычисление коэффициента обнаружения мутантов для всей кодовой базы Google остается непомерно дорогим из-за ее размера. Было бы еще дороже продолжать пересчитывать коэффициент обнаружения мутантов, например, ежедневно или еженедельно, и невозможно вычислить его после каждого фиксации. Помимо затрат на вычисление этого коэффициента, мы не смогли найти хороший способ сообщить о нем разработчикам в действенной форме: он не является ни конкретным, ни действенным, и не служит руководством для тестирования. Сообщать разработчикам об отдельных мутантах в больших масштабах также сложно, особенно из-за непродуктивных мутантов.

Решая проблему масштабных и непродуктивных мутантов, мы разработали и внедрили подход к тестированию мутаций, который отличается от традиционного подхода, описанного в литературе [14]. В целях масштабируемости мы разработали и внедрили тестирование мутаций на основе различий, которое генерирует и оценивает мутанты только для покрытых измененных линий; для повышения продуктивности мы разработали и внедрили подход к подавлению мутантов и вероятностному отбору мутантов. Мутационное тестирование в Google начинается, когда разработчик отправляет изменение кода на проверку кода. Процесс тестирования мутаций состоит из четырех этапов высокого уровня: анализ покрытия кода (раздел 2.1), генерация мутантов (раздел 2.2), анализ мутаций (раздел 2.3) и отчет о выживших мутантах в процессе проверки кода (раздел 2.4).

На рис. 1 подробно описана служба мутационного тестирования. (1) Все начинается со списка изменений, отправленного на проверку кода. (2) Как только метаданные покрытия кода становятся доступными, они определяют набор строк, которые покрываются, а также добавляются или изменяются в списке изменений. (3) Затем он создает AST для каждого затронутого файла и посещает каждый защищенный узел. (4) Затем он помечает засушливые узлы (узлы, которые в случае мутации создают непродуктивных мутантов) на основе эвристики, накопленной с использованием отзывов разработчиков о продуктивности мутантов на протяжении многих лет. Маркировка засушливых узлов происходит до того, как генерируются мутанты, и, следовательно, мутанты в засушливых узлах вообще никогда не генерируются. (5) Затем мутагенез генерирует мутанты для подходящих узлов (т.е. каждого узла, который не является засушливым и охвачен по крайней мере одним тестом). (6) Служба мутационного тестирования затем оценивает мутанты по существующим тестам и (7) сообщает о подмножестве выживших мутантов в качестве результатов проверки кода.





**2.1 Предварительные условия: списки изменений и покрытие**

Список изменений представляет собой атомарное обновление системы контроля версий и состоит из списка файлов, операций, которые необходимо выполнить над этими файлами, и, возможно, содержимого файла, которое необходимо изменить или добавить, а также метаданных, таких как описание изменения, автор, и т. д.

После того как разработчик отправляет список изменений коллегам-разработчикам для проверки кода, для этого списка изменений выполняются различные статические и динамические анализы, а результаты сообщаются разработчику и рецензентам. Покрытие строк — один из таких анализов: во время проверки кода разработчикам сообщается общее и дельта-покрытие кода [1]. Общее покрытие кода — это отношение количества строк, покрытых тестами в файле, к общему количеству инструментированных строк в файле. Количество инструментированных строк обычно меньше общего количества строк, поскольку такие артефакты, как комментарии или строки с чистыми пробелами, исключаются. Дельта-покрытие — это отношение количества охваченных добавленных или измененных строк к общему количеству добавленных или измененных строк в списке изменений. На рисунке 2 показано распределение покрытия линий по проектам, что указывает на то, что покрытие линий для большинства проектов является удовлетворительным.

Покрытие кода является обязательным условием для проведения анализа мутаций из-за высокой стоимости создания и оценки мутантов в непокрытых линиях, которые неизбежно выживут, поскольку код не тестируется. Как только для списка изменений становится доступным покрытие на уровне строки, запускается мутагенез. Google использует Bazel в качестве своей системы сборки [15]. Цели сборки явно перечисляют свои источники и зависимости и соответствуют произвольному количеству целей тестирования, каждая из которых может включать несколько тестов. Тесты выполняются параллельно. Используя явные зависимости и список исходных кодов, анализ покрытия кода предоставляет информацию о том, какая цель теста охватывает какие строки исходного кода, тем самым связывая строки кода с набором покрывающих их тестов. Покрытие на уровне линии используется для определения набора тестов, которые необходимо провести, чтобы попытаться убить мутанта. Этот подход также реализован в других инструментах тестирования мутаций, включая PIT [16] и Major [17], [18].

**2.2 Мутагенез**

В службу мутагенеза поступает запрос на создание точечных мутаций, то есть мутаций, в результате которых образуется мутант, отличающийся от оригинала в одном узле AST на запрошенной линии. Для каждого поддерживаемого языка программирования специальная служба мутагенеза, способная перемещаться по AST модуля компиляции на этом языке, принимает запросы на точечную мутацию и отвечает потенциальными мутантами. Операторы мутации реализованы как посетители AST, этот подход также используется в других инструментах мутации (например, [19]). Для каждого запроса точечной мутации, т. е . кортежа (файл; строка) , выбирается оператор мутации, и в этой строке генерируется мутант, если этот оператор мутации к нему применим. Если оператор мутации не генерирует мутанта, выбирается другой оператор и так далее, пока либо мутант не будет сгенерирован, либо все операторы мутации не будут опробованы и мутант не может быть сгенерирован. Существует две стратегии выбора оператора мутации: случайная и целевая, подробно описанные в разделе 4. Служба тестирования мутаций генерирует не более одного мутанта на линию из соображений масштабируемости и на основе понимания того, что подавляющее большинство мутантов для данной линии имеют одинаковые судьба — либо все, либо ни один из них не выживет в результате анализа [20]. Это означает, что если мутант, полученный для данной линии, не выдерживает мутационного анализа, для этой линии не создается никаких дополнительных мутантов. Служба мутационного тестирования реализует мутагенез для 10 языков программирования: C++, Java, Go, Python, TypeScript, JavaScript, Dart, SQL, Common Lisp и Kotlin. Для каждого языка служба реализует пять операторов мутации: AOR (замена арифметического оператора), LCR (замена логического соединителя), ROR (замена реляционного оператора), UOI (вставка унарного оператора) и SBR (удаление блока операторов). Эти операторы мутации были первоначально введены для Мотры [21], и в Таблице 1 приведены примеры для каждого из них. В Python унарное увеличение и уменьшение заменяются бинарным оператором для достижения того же эффекта благодаря конструкции языка. По нашему опыту, оператор мутации ABS (вставка абсолютного значения) преимущественно создавал непродуктивные мутанты, главным образом потому, что он действовал на выражения, связанные со временем и счетом, которые являются положительными и бессмысленными, если их отрицать. Поэтому служба мутационного тестирования не использует оператор ABS. Обратите внимание, что наши наблюдения в целом могут не соответствовать действительности и могут зависеть от стиля и особенностей нашей кодовой базы.

**2.3 Анализ мутаций**

После того как мутагенез генерирует набор мутантов для списка изменений, для каждого из них подготавливается временное состояние системы контроля версий на основе исходного списка изменений, а затем для всех этих состояний параллельно выполняются тесты. Это обеспечивает эффективное взаимодействие и кэширование между нашей системой контроля версий и системой сборки, а также позволяет максимально быстро оценивать мутанты.

Как только результаты анализа мутаций становятся доступными, Служба тестирования мутаций выбирает и сообщает о мутантах из набора выживших мутантов. Мы ограничиваем количество зарегистрированных мутантов максимум в 7 раз больше общего количества файлов в списке изменений. Это гарантирует, что когнитивные издержки, связанные с пониманием всех зарегистрированных мутантов, не будут слишком высокими, что в противном случае может привести к тому, что разработчики перестанут использовать тестирование мутаций. Мы эмпирически определили, что 7 — это подходящий компромисс между эффективностью теста и когнитивной нагрузкой, собирая данные за годы работы системы. Наконец, служба сообщает автору и рецензентам о выбранных выживших мутантах в пользовательском интерфейсе проверки кода. Обратите внимание, что для обеспечения единообразия служба мутационного тестирования выбирает и сообщает о мутантах в той же строке (строках), что и раньше, если автор добавляет дополнительные тесты или иным образом обновляет список изменений, что запускает повторное выполнение службы.

**2.4 Сообщение о мутантах в процессе проверки кода**

Большинство изменений в кодовой базе Google, за исключением ограниченного

количество полностью автоматизированных изменений, проверенных разработчиками

прежде чем они будут объединены в дерево исходного кода. Потвин

и Левенберг [9] дают исчерпывающий обзор

Экосистема разработки Google. Рецензенты могут оставлять комментарии

по измененному коду, который должен разрешить автор.

Особый тип комментария, генерируемый автоматическим

анализатор называется находкой. В отличие от созданного человеком

комментарии, выводы не требуют разрешения автора

перед отправкой, если только рецензент не пометит их как

обязательный. Многие анализаторы запускаются автоматически, когда

список изменений отправлен на рассмотрение: линтеры, форматтеры, статический код

и строить анализаторы зависимостей и т. д. Большинство анализаторов

основаны на платформе анализа кода Tricorder [22].

Служба мутационного тестирования сообщает об избранных мутантах.

разработчикам в процессе проверки кода, что максимизирует

есть вероятность, что они будут учтены разработчиками.

Количество комментариев, отображаемых во время кода

обзор может быть большим, поэтому важно, чтобы все инструменты производили

практические результаты, которые могут быть немедленно использованы

Разработчики. Отчет о недействительных выводах во время написания кода

рецензия оказывает негативное влияние на автора и рецензентов.

Если находка (например, выживший мутант) не воспринимается

если это полезно, разработчики могут сообщить об этом одним щелчком мыши по

находка. Если кто-либо из рецензентов считает вывод

важно, они могут указать это автору списка изменений

одним щелчком мыши. На рисунке 3 показан пример мутанта.

отображается в Critique, системе проверки кода Google [23],

включая ссылки «Пожалуйста, исправьте» и «Бесполезно» в

нижние углы. Этот отзыв доступен владельцу

систему, которая создала результаты, поэтому показатели качества могут

отслеживаться, а результаты, не требующие принятия мер, сортироваться и, в идеале,

предотвращено в будущем.

Чтобы быть полезным автору и рецензентам, код

выводы должны быть действенными и сообщаться быстро, прежде чем

обзор завершен. С этой целью мутационное тестирование

Служба выполняет подавление мутантов (раздел 3), и это

вероятностно отбирает мутантов на основе их исторического прошлого.

производительность оператора мутации (раздел 4).

**3. ПОДАВЛЕНИЕ НЕПРОДУКТИВНЫХ МУТАНТОВ​​**

Некоторые части кода менее интересны, чем другие.

Сообщение о живых мутантах в неинтересных утверждениях (например,

операторы протоколирования для целей отладки) имеет отрицательный результат

влияние на когнитивную нагрузку и время, затрачиваемое на анализ мутантов.

Потому что разработчики не воспринимают добавление тестов как убийство.

мутанты в неинтересном коде как улучшение общего

эффективности набора тестов, такие мутанты имеют тенденцию выживать и

быть помечены как непродуктивные.

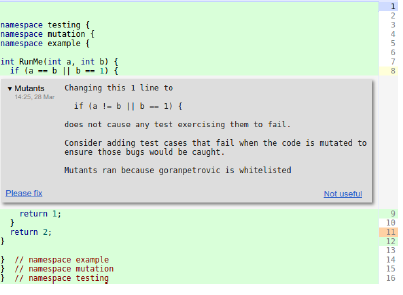


Рис. 3. Сообщение Mutant в инструменте проверки кода.

В этом разделе предлагается подход к подавлению непродуктивных

мутантов, основанный на наборе эвристик для обнаружения

засушливые (т.е. неинтересные) узлы AST. Существует компромисс

между корректностью и удобством использования результатов; эвристика

может предотвратить мутацию в очень небольшом количестве незасушливых узлов, поскольку

побочный эффект подавления мутаций во многих засушливых узлах.

Мы утверждаем, что это хороший компромисс, поскольку число

возможных мутантов на порядки больше, чем

что служба мутаций могла бы разумно сообщить

разработчиков в рамках существующих инструментов разработчика. Более того,

предотвращение выводов, не имеющих практического значения, важнее, чем

сообщать обо всех практических выводах.

**3.1 Обнаружение засушливых узлов**

Чтобы предотвратить появление непродуктивных мутантов,

Служба тестирования мутаций идентифицирует засушливые узлы

в АСТ, которые относятся к неинтересным высказываниям.

Примеры засушливых узлов включают вызовы резервирования памяти.

такие функции, как std::vector::reserve и запись в стандартный вывод;

они обычно не проверяются модульными тестами.

Операторы мутации создают мутантов на основе AST

программа. AST содержит узлы, которые являются операторами,

выражения или декларации, а также их дочерние и родительские отношения

отразить их связи в исходном коде [24]. Большинство

компиляторы различают простой и составной AST

узлы. Простые узлы не имеют тела; например, вызов функции

выражение предоставляет имя функции и аргументы,

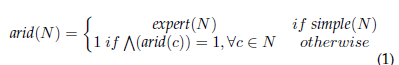
но не имеет тела. Составные узлы имеют хотя бы одно тело;

например, цикл for может иметь одно тело, а цикл if

Оператор может иметь две ветви — then и else.

Наш основанный на эвристике подход к маркировке узлов как засушливых

является двойным:



Здесь N 2 T — узел абстрактного синтаксического дерева T

программа, simple — это булева функция, определяющая, будет ли

узел — простой или составной узел (составные узлы содержат

их дочерние узлы c ), а эксперт является частичным логическим значением

функция, отображающая подмножество простых узлов в T на свойство быть засушливым. Первая часть уравнения 1 действует

на простых узлах, используя экспертную функцию, которая кодирует

знания, которые вручную отбираются для каждого программирования

язык и корректировались с течением времени. Вторая часть действует

на составных узлах и определяется рекурсивно. Соединение

узел засушлив тогда и только тогда, когда все его дочерние узлы засушливы.

Экспертная функция помечает простые узлы как засушливые и

на основе отзывов разработчиков о сообщении «Бесполезно»

мутанты. Это ручной процесс: если мы определим, что

определенный мутант действительно непродуктивен, и что целый

класс таких мутантов создавать не следует, добавлено правило

к экспертной функции. Это ключевой компонент

Служба мутационного тестирования — без нее пользователи стали бы

разочарованы выводами, не имеющими практического значения, и отказываются от участия

система в целом. Целенаправленная мутация и тщательная отчетность

мутантов сыграли решающую роль в принятии мутации

тестирование в Google. На данный момент мы накопили более

сто правил для обнаружения засушливых узлов.

**3.2 Экспертные эвристические категории**

Экспертная функция состоит из различных правил, некоторые из которых

специфичны для оператора мутации, а некоторые из них универсальны.

Мы различаем эвристики, которые предотвращают

генерация некомпилируемых и компилируемых, но непродуктивных

мутанты. Большинство эвристик имеют дело с последней категорией, но

первое также важно, особенно в Го, где

компилятор очень чувствителен к мутациям (например, неиспользуемый

импорт — ошибка компилятора). Для компилируемых мутантов мы

далее различать эвристики для эквивалентных мутантов,

убиваемые мутанты и избыточные мутанты, как сообщается

в таблице 2.

Каждая из четырех эвристических категорий содержит одну или

более отдельные группы правил, которые, в свою очередь, содержат один

или несколько связанных правил. Например, все правила, подавляющие

мутанты в операторах регистрации (несколько правил для нескольких

типы операторов и функций журналирования) образуют отдельный

группу, поскольку все они применимы к ведению журнала, и вся

Целью группы является предотвращение появления непродуктивных убиваемых мутантов.

частота показывает, как часто категория применима к

данный список изменений. Подробный список правил см.

дополнительные материалы, которые можно найти в Интернете по адресу

<производственный персонал вставит ссылку>.

3.2.1 Эвристика для предотвращения некомпилируемых мутантов

Мутант должен быть синтаксически допустимой программой.

в противном случае он будет обнаружен компилятором и будет

не добавляет никакой ценности для тестирования. Существуют определенные мутации,

особенно те, которые удаляют код, нарушающий эту действительность

принцип. Яркий пример — удаление кода в Go; любой

неиспользуемая переменная или импортированный модуль создает компилятор

ошибка. Предлагаемая эвристика собирает все используемые символы и

помещает их в контейнер, а не удаляет, чтобы они

остаются ссылки, и компилятор успокаивается.

3.2.2 Эвристика для предотвращения эквивалентных мутантов

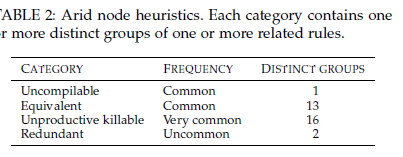
Эквивалентные мутанты, семантически эквивалентные

мутировавшая программа - это чума в тестировании мутаций

и обычно не могут быть обнаружены автоматически. Однако,

существуют некоторые группы эквивалентных мутантов, которые могут быть

точно обнаружено. Например, в Java спецификация



для метода размера java.util.Collection заключается в том, что он

возвращает неотрицательное значение. Это означает, что мутации

например, Collection.size() == 0 7! коллекция.размер() <= 0

гарантированно дадут эквивалентного мутанта.

Другой пример этой категории связан с мемоизацией.

Мемоизация часто используется для ускорения выполнения, но

его удаление неизбежно приводит к образованию эквивалентного

мутанты. Для обнаружения мемоизации используется следующая эвристика:

оператор if является поиском в кеше, если он имеет форму if

а, ок := x[v]; ок, верните a, т. е. при поиске на карте

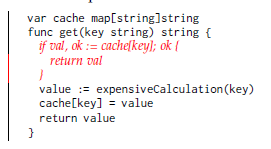
находит элемент, блок if возвращает этот элемент (среди

другие значения, например, Ошибка в Go). Такой оператор if является

оператор поиска в кэше и эксперт считает его пустым

функционировать, как и все его тело. В следующем примере показан

поиск в кеше в Go:



Удаление оператора if просто удаляет кеширование, но не

не меняет функциональное поведение и, следовательно, дает эквивалент

мутант. Программа по-прежнему выдает тот же результат

для того же ввода, хотя и медленнее. Функциональные тесты не

ожидалось обнаружить такие изменения.

В качестве третьего примера: эвристика в этой категории позволяет избежать

мутации спецификаций времени, поскольку модульные тесты редко проверяют

за временем, а если и делают, то, как правило, используют поддельные часы. Заявления

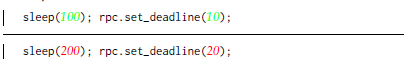
вызов функций, подобных сну, установление сроков или

ожидание готовности сервисов (например, сервера gRPC [25])

Функция ожидания, которая всегда вызывается на серверах RPC, что

изобилуют кодовой базой Google), считаются засушливыми

экспертная функция.



3.2.3 Эвристика для предотвращения непродуктивных убиваемых мутантов

Не весь код одинаково важен: некоторый код может привести к

убиваемые мутанты, но тесты, которые их убивают, бесполезны

и не будет написан опытными разработчиками; такой

мутанты — плохие цели для испытаний. Примеры этой категории:

прирост значений в рамках системы мониторинга, низкий

API-интерфейсы уровня или изменения флагов: их легко изменить, легко

тест для, но в основном нежелательных целей тестирования.

Распространенный способ реализации эвристики в этой категории.

заключается в сопоставлении имен функций; действительно, мы подавляем мутантов в

вызовы сотен функций, на которые приходится большая часть подавлений экспертной функцией.

Ярким примером этой категории является эвристика, которая отмечает

любой вызов функции, если имя функции начинается с

журнал префикса или объект, для которого вызывается функция,

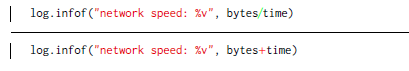
называется логгером. Мы проверили эту эвристику путем случайной выборки.

100 узлов, отмеченных эвристикой журнала,

и обнаружил, что 99 действительно были отмечены правильно, а один

имел предельную полезность. Итого у нас накопились нечеткие

правила подавления имен для более чем 200 семейств функций.



3.2.4 Эвристика для предотвращения избыточных мутантов

Напомним, что Служба мутационного тестирования генерирует не более

по одному мутанту на линию и сообщает об ограниченном подмножестве

выжившие мутанты во время проверки кода. Эвристика в этом

категория подавляет некоторые избыточные мутанты (т.е.

функционально эквивалентен другим мутантам) по двум причинам.

Во-первых, хотя избыточные мутанты функционально эквивалентны

друг другу, о некоторых из них легче рассуждать

чем другие, что делает их более продуктивными. Второй,

когда разработчик обновляет свой список изменений, возможно, написав

тесты на уничтожение мутантов, это изменение создает новый снимок

и запускает повторный запуск службы мутации, тем самым проверяя

изменение и, возможно, сообщение о новых мутантах. Чтобы

для повышения производительности разработчиков и удобства пользователей,

Служба тестирования мутаций должна последовательно генерировать

того же мутанта из пула одинаково продуктивных и

избегать расхождений с ранее зарегистрированными мутантами, в частности

для неизмененных строк между снимками. Такое расхождение

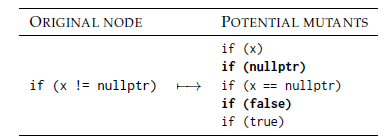
вызовет путаницу, приведет к когнитивным издержкам,

и, следовательно, более низкая производительность разработчиков.

Например, в C++ оператор мутации LCR имеет

особый случай при работе с NULL (т. е. nullptr), поскольку

его логической эквивалентности ложному:



Мутанты, выделенные жирным шрифтом, являются избыточными, поскольку

значение nullptr эквивалентно false . Аналогично, наоборот

пример, где условие if (nullptr == x),

дает избыточные мутанты для левой части.

3.2.5 Опыт работы с эвристикой

По нашему опыту применения эвристики, наибольшая производительность

прибыль стала результатом внедрения трех эвристик

в первые дни: подавление мутаций в журналировании

заявления, операции, связанные со временем (например, установление сроков,

таймауты, характеристики экспоненциальной задержки и т. д.), и, наконец,

флаги конфигурации. Большая часть ранних отзывов касалась

непродуктивные мутанты в таком коде, который повсеместно встречается в

кодовая база. Хотя это трудно измерить точно, существует

является убедительным свидетельством того, что эти подавления являются причиной

повышение производительности примерно с 15% до 80%. Дополнительный

эвристика и уточнения постепенно улучшаются

производительность до 89%.

Эвристика реализуется путем сопоставления узлов AST.

с полной информацией компилятора, доступной для мутации

оператор. Некоторые эвристики несостоятельны: они используют

нечеткое сопоставление имен и распознавание форм AST, но может

подавлять продуктивных мутантов. С другой стороны, некоторые

эвристики используют полную информацию о типе (например, сопоставление

java.util.HashMap::size) и являются надежными. Звук

эвристики явно верны, но у нас было много

более важные улучшения воспринимаемой полезности мутантов

от необоснованной эвристики.

**4 СТРАТЕГИИ ВЫБОРА ОПЕРАТОРА МУТАЦИИ​​​**

После маркировки засушливых узлов в AST, тестирование мутаций

Сервис генерирует мутантов для оставшихся, не засушливых

узлы. Это предполагает две проблемы. Во-первых, генерируется только

о мутантах, выживших в испытаниях, сообщается разработчикам

во время проверки кода; мутанты, которые не выживают, просто используют

вычислительные ресурсы. Учитывая, что многие мутанты не

выжить в тестах, а мутагенез генерирует только один

мутанта на линию, цель состоит в том, чтобы создать мутантов, которые имеют

высокие шансы на выживание. Итеративный подход, при котором после

первый раунд испытаний, дальнейшие раунды мутагенеза могут

для линий, в которых были убиты мутанты, будет использоваться

создавать и тестировать системы неэффективно и потребует много

дольше из-за нескольких раундов. Аналогично, генерируя все

мутантов на линию — это слишком дорого в вычислительном отношении. Второй,

не все выжившие мутанты одинаково продуктивны: в зависимости от

в контексте некоторые операторы мутации могут создавать

лучшие мутанты, чем другие. Поэтому цель состоит в том, чтобы создать

выжившие мутанты, имеющие высокие шансы на продуктивность.

Эффективная стратегия выбора оператора мутации не

представляет собой лишь хороший компромисс между производительностью и

затрат, но также имеет решающее значение для получения результатов анализа мутаций.

применимо во время проверки кода.

В этом разделе представлена базовая стратегия случайного выбора.

что генерирует одного мутанта на покрытую линию, учитывая

информация об засушливых узлах и стратегия целевого отбора,

который дополнительно учитывает прошлые результаты

операторы мутации в аналогичном контексте (рис. 4).

**4.1 Случайный выбор**

Базовый подход к отбору мутантов на основе случайных линий мог бы:

для каждой строки в списке изменений выберите одного из мутантов

который может быть сгенерирован для этой линии равномерно случайным образом.

Альтернативно, такой подход может случайным образом выбирать

сначала точка мутации в этой строке, а затем случайным образом выбранная

применимый оператор мутации.

Напомним, что наш подход к тестированию мутаций основан

по выявлению засушливых узлов, которых не следует

вообще мутировал. Более того, наш подход генерирует максимум

один мутант на линию; никаких дополнительных мутантов никогда не образуется.

В листинге 1 описан наш алгоритм случайного выбора.

это объясняет эти два проектных решения. Мутация

операторы, доступные для данного языка, перемешиваются случайным образом

и пробовал по одному, для каждой пройденной, измененной строки

соответствующие незасушливым узлам в списке изменений, пока не

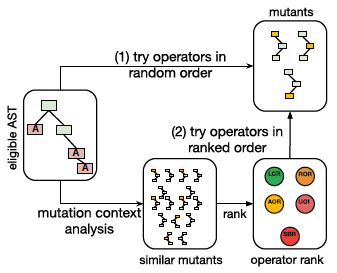


Рис. 4: Случайный (1) и целевой (2) выбор мутаций.

для этой строки создан мутант или все операторы были

пытался. Если в линии может быть создано несколько мутантов, то только

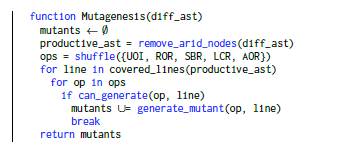
генерируется один мутант, но какой именно, зависит от

случайное перемешивание и сам AST. Например, РОР

оператор мутации не может сгенерировать мутанта в строке, которая

не имеет реляционных операторов, но оператор SBR может —

большинство строк можно удалить.



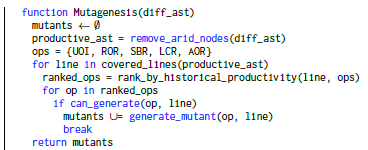
Листинг 1. Случайный выбор с подавлением.

**4.2 Целевой отбор**

В отличие от случайного отбора, целевой отбор

стратегия ранжирует операторы мутации по их исторической продуктивности

учитывая контекст AST, как показано в листинге 2.



Листинг 2: Целевой отбор с подавлением.

Рейтинг оператора мутации для данного узла AST равен

на основе исторической информации, в частности живучести

и производительность. Выживаемость оператора мутации – это

соотношение выживших мутантов, созданных этим оператором в

данный контекст. Производительность оператора мутации равна

соотношение продуктивных мутантов, созданных этим оператором в

данный контекст. Производительность основана на отзывах разработчиков:

во время проверки кода авторы и рецензенты могут отмечать мутанты

отображается в списке изменений как продуктивный или непродуктивный. Как

эти разработчики понимают контекст мутантов, которых они

помечают, в отличие от участников, выполняющих задачу по маркировке

в исследовании мы считаем эту информацию сильным сигналом.

Для каждого мутанта контекст AST, описывающий

среда узла AST, который был мутирован, сохраняется

наряду с отзывами о продуктивности и о том, является ли мутант

был убит или нет. Стратегия целевого отбора использует

эту информацию для идентификации узлов AST, похожих на

мутированный, основанный на контексте AST. Исторический

информация о мутантах, созданных для этих подобных AST

узлов затем используется для ранжирования операторов мутации, а не

чем использовать случайный порядок. Затем предпринимают попытку мутагенеза.

в полученном порядке, чтобы максимизировать вероятность того, что

мутант выживет и будет продуктивным.

**4.3 Контекст мутации**

В целях применения исторических сведений о живучести

и продуктивности, нам нужно решить, насколько похожий кандидат

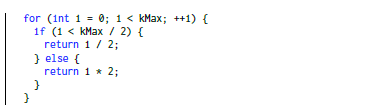
мутации сравниваются с прошлыми мутациями. Мы определяем

мутация будет похожей, если она произошла в аналогичном контексте,

например, замена оператора отношения в условии if

это первый оператор в теле цикла for, как показано

в листинге.



Листинг 3. Фрагмент кода C++: оператор if внутри цикла for.

Чтобы эффективно уловить сходство контекста

две мутации, мы используем структуру хеширования для древовидной структуры

данные, представленные Татикондой и др. [26], в котором

отображает неупорядоченное дерево в мультимножество простых структур

называемые поворотными точками . Каждый поворот фиксирует информацию

о взаимоотношениях между узлами дерева (см.

раздел 4.4).

Поиск подобных контекстов мутации затем сводится к

поиск похожих опорных мультимножеств. Чтобы определить похожий центр

мультимножества, мы создаем отпечаток пальца, вдохновленный MinHash [27]

поворотного мультисета. Потому что расстояние в отпечатке пальца

пространство коррелирует с расстоянием в древовидном пространстве, мы можем

эффективно находить похожие контексты мутаций, находя похожие

отпечатки узлов, подвергшихся мутации.

**4.4. Генерация опорных точек на основе AST**

Чтобы уловить сложные отношения между узлами

в AST мы переводим AST в мультинабор опорных точек. А

Pivot — это тройка узлов из AST, которая кодирует их

отношение; для узлов u и v стержнем p является кортеж ( lca; u; v) ,

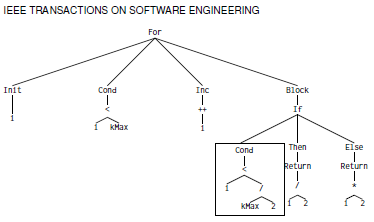
где lca — младший общий предок узлов u и

Рис. 5: AST для примера C++ в листинге 3.

*в .* Сводная точка представляет собой поддерево AST. Набор

все повороты, включающие определенный узел, описывают дерево

с точки зрения этого узла. При мутационном тестировании

нас интересуют только узлы, находящиеся близко к узлу

мутируется, поэтому мы ограничиваем набор опорных точек

содержащие узлы, находящиеся на определенном расстоянии от узла

рассматривается на предмет мутации.

В примере замены оператора отношения в

условие if в теле цикла for в листинге 3, один

точка поворота может быть ( if, Cond, \_) и другая ( Cond, i , kMax ) . Все

комбинации двух узлов на некотором расстоянии от

узел, мутирующий в AST на рисунке 5, и их самый низкий

общего предка составляют опорные конструкции.

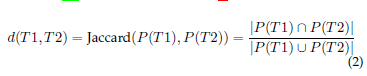
Поворотные мультимножества P точно сохраняют структурные отношения.

узлов дерева (отношения родитель-потомок и предок),

поэтому сходство двух поддеревьев AST T1 и

T2 можно измерить как индекс Жаккара опорной точки.

мультимножества [26], как показано в уравнении 2.



**4.5 Мультимножества Pivot для снятия отпечатков пальцев**

Сводные мультимножества потенциально квадратичны по размеру дерева, что приводит к

дорогостоящим операциям объединения и пересечения. Даже тривиальное

оператор if с одним оператором возврата создает большие

сводные наборы и операции с множествами становятся непомерно трудными. Облегчить

что функция снятия отпечатков пальцев применяется для преобразования больших

сводные мультинаборы в отпечатки пальцев фиксированного размера.

Мы хэшируем сводные наборы с отдельными объектами, образующими

мультимножество представителей для входного AST. Размер

multiset может быть большим, особенно для больших программ. Чтобы

для повышения эффективности дальнейших манипуляций мы используем

сигнатурная функция, которая преобразует большие наборы сводных хэшей в

короче подписи. Сигнатуры позже используются для вычисления

сходство между деревьями, принимая во внимание

только тип узла AST и игнорируя все остальное, например

введите данные или имена идентификаторов.

Мы используем простую хэш-функцию для хеширования одной опорной точки *p =*

*(lca; u; v)* в фиксированную величину, предложенную Татикондой и

Партасаратхи [26].



Для ; б; c мы выбираем маленькие простые числа, а для K — большое простое число

это умещается в 32 бита. Чтобы иметь возможность хешировать узлы AST, мы назначаем

разреженные целочисленные хеш-значения для разных типов узлов AST в

каждому языку, например, FunctionDecl C++ присваивается значение 8500,

и CXXMethodDecl 8600. Для узлов в центре (lca; u; v)

мы используем эти назначенные хеши.

Например, учитывая a = 17 , b = 59 , c = 83 и K =

15485863 , мы можем вычислить хеш опорной точки ( if, <, \_) ,

так же просто, как



где 32800 и 22400 являются целочисленными хэш-значениями

назначен узлам IfStmt и BinaryOperator C++ AST.

Подпись для такого пакета представителей генерируется

с использованием техники MinHashing. Набор шарниров есть

перестановлен и хеширован под этой перестановкой. Чтобы свести к минимуму

ложные срабатывания и негативы (т. е. разные деревья хеша

на аналогичные хэши или наоборот), это повторяется k раз,

в результате получается k -MinHashes.

Цель состоит в том, чтобы подписи были похожи для похожих

(мульти)множества и непохожие для непохожих. Сходство Жаккара

между двумя наборами можно оценить путем сравнения

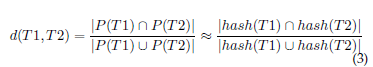
их подписи MinHash таким же образом [27], как показано

в уравнении 3. Схему MinHash можно рассматривать как

пример хеширования с учетом местоположения, при котором AST, которые

находящиеся на небольшом расстоянии друг от друга, превращаются в

хеши, которые сохраняют это свойство.



При мутации узла мы вычисляем его набор узлов и

хешируйте его. Мы находим похожие контексты AST, используя ближайшего соседа.

алгоритмы поиска. Наблюдаем, как ведут себя разные мутанты

в этом контексте и какие операторы мутации производят

самые продуктивные и выживающие мутанты. Это основа

для целевого отбора мутаций.

**5 ОЦЕНКА**

Чтобы принести пользу разработчикам, Mutation Testing

Службе Google необходимо сообщить о нескольких продуктивных мутантах.

выбраны из большого пула мутантов, большинство из которых

являются непродуктивными. Напомним, что продуктивный мутант вызывает

эффективный тест или иным образом повышает качество кода [11].

Поэтому наша цель двоякая. Во-первых, мы стремимся отобрать мутантов

с высокой выживаемостью и продуктивностью для максимизации

их полезность в качестве целей тестирования. Во-вторых, мы стремимся сообщить

очень мало мутантов, чтобы уменьшить вычислительные усилия и избежать

ошеломляющие разработчиков слишком большим количеством выводов.

Поскольку применение мутационного тестирования ко всей базе кода

просто неосуществимо, мы сосредотачиваемся на мутации на основе различий в наших

оценка. Помимо основного конструктивного решения применения

мутационное тестирование на уровне списков изменений, два технических

решения уменьшают количество мутантов: (1) мутант

подавление с использованием засушливых узлов и (2) мутанта по одному на линию

выбор. В нашей оценке используются два набора данных (раздел 5.1) и

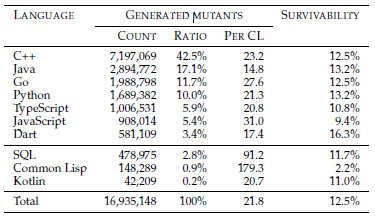
отвечает на четыре исследовательских вопроса. Первый исследовательский вопрос

касается эффективности двух наших технических решений:

ТАБЛИЦА 3: Краткое описание набора данных о мутантах. (Обратите внимание, что SQL,

Common Lisp и Kotlin исключены из нашего анализа.

из-за недостаточности данных.)



\_ **RQ1 Подавление мутантов** . Насколько эффективен мутант

подавление с использованием засушливых узлов и мутанта 1 на линию

выбор? (раздел 5.2)

Понять влияние выбора оператора мутации

на выживаемость и продуктивность мутантов в остальных

неаридные узлы, мы рассматриваем исторические данные, в том числе

отзывы разработчиков. Мы стремимся ответить на следующие два

исследовательские вопросы:

\_ **RQ2 Живучесть мутанта** . Оператор мутации

отбор влияет на вероятность того, что сгенерированный мутант

выдержит набор тестов? (раздел 5.3)

\_ **RQ3 Продуктивность мутантов** . Оператор мутации

выбор влияет на отзывы разработчиков о сгенерированном

мутант? (раздел 5.4)

Установив влияние отдельных операторов мутации

по живучести и производительности, последний вопрос

заключается в том, можно ли использовать контекст мутации для улучшения обоих.

Таким образом, наш последний исследовательский вопрос заключается в следующем:

\_ **Контекст мутации RQ4** . Осуществляет ли контекстный выбор

операторы мутации улучшают выживаемость мутантов и

производительность? (раздел 5.5)

**5.1. Постановка эксперимента**

Для нашего анализа мы создали два набора данных, один с

данные обо всех мутантах, а один содержит дополнительные данные о

контекст мутации для подмножества всех мутантов.

**Набор данных мутантов.** Набор данных мутантов содержит 16 935 148

мутанты по 10 языкам программирования: C++, Java, Go,

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

и Котлин. В Таблице 3 обобщен набор данных о мутантах и

дает количество и соотношение мутантов на одно программирование

язык, среднее количество мутантов в списке изменений

и процент мутантов, выживших в серии тестов.

В таблице 4 числа разбиты по операторам мутации.

Мы создали этот набор данных, собрав данные обо всех мутантах.

которые Служба тестирования мутаций создала с момента своего

инаугурация, которая относится к дате, когда мы сделали

услуга широко доступна после первоначальной разработки

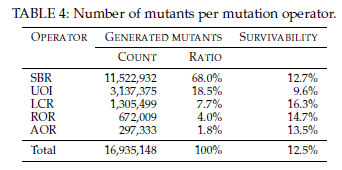
сервис и правила его подавления (см. раздел 3.2.5). Мы сделали

не выполнять никакой фильтрации данных, следовательно, набор данных обеспечивает

информацию обо всех проведенных мутационных анализах.

Всего в нашем сборе данных было рассмотрено 776 740 списков изменений.

которые были частью процесса проверки кода. Для этих,



Было создано 16 935 148 мутантов, из них 2 110 489

были сообщены. Из всех зарегистрированных мутантов 66 798 получили

явная обратная связь с разработчиком. Для каждого рассматриваемого списка изменений

набор данных мутантов содержит информацию о:

\_ затронутые файлы и затронутые строки,

\_ тестовые цели для тестирования затронутых строк,

\_ мутанты, созданные для каждой из затронутых линий,

\_ результаты тестирования файла в измененной строке и

\_ оператор мутации и контекст для каждого мутанта.

Наш анализ направлен на изучение эффективности и воспринимаемых

продуктивность мутантов и операторов мутаций в программировании

языки. Обратите внимание, что наш набор данных мутантов, скорее всего,

специфичны для стиля кода Google и практики проверки. Однако,

стиль кода получил широкое распространение [28], а современный

Процесс проверки кода используется во всей отрасли [29].

Информация о выживаемости мутантов за программу

оператор языка или мутации может быть извлечен напрямую

из набора данных и позволяет нам отвечать на исследования

вопросы **RQ1** , **RQ2** и **RQ3** .

**Контекстный набор данных.** Набор контекстных данных содержит 4 068 241

мутанты (подмножество набора данных мутантов) для четырех лучших

языки программирования: C++, Java, Go и Python. Каждый

мутант в этом наборе данных обогащен информацией

будет ли наша стратегия выбора на основе контекста иметь

выбрал этого мутанта. При создании мутантов мы бы

также запустили контекстно-ориентированное прогнозирование, и мы настояли на этом

информацию предсказания вместе с мутантами. Если

случайно выбранный оператор действительно оправдал предсказание

выбранной службой, у этого мутанта самый высокий

прогнозируемое значение. Для каждого мутанта набор данных содержит:

\_ вся информация из набора данных мутантов,

\_ прогнозируемая выживаемость и продуктивность для каждой мутации

в аналогичном контексте и

\_ информация о том, имеет ли мутант высший

прогнозируемая живучесть/производительность.

Мы создали этот набор данных, используя нашу контекстную мутацию.

стратегия отбора при мутагенезе всех мутантов

в течение ограниченного периода времени. За это время мы

автоматически аннотировал мутантов, указывая, есть ли

мутант будет выбран с помощью контекстной мутации

стратегия отбора наряду с результатом мутанта с точки зрения

живучести и продуктивности. Этот набор данных позволяет

оценка нашей стратегии выбора мутаций на основе контекста

и позволяет нам ответить на исследовательский вопрос **RQ4** .

**Меры эксперимента:** Выживание в первоначальном наборе тестов — это

предварительное условие для появления мутанта, но только выживаемость

не является хорошим показателем продуктивности мутантов. Разработчик

обратная связь, показывающая, что мутант действительно (не)продуктивен

это более сильный сигнал.

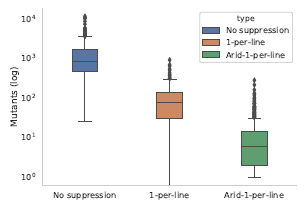


Рис. 6: Количество сгенерированных мутантов в списке изменений для no

подавление (традиционный мутагенез), 1 на линию и арид-1-

построчно (наш подход). (Обратите внимание на вертикальную ось в логарифмическом масштабе.)

Мы измеряем продуктивность мутантов на основе отзывов пользователей.

из Критики (раздел 2.4), где каждый заявленный мутант

отображает «Пожалуйста, исправьте» (продуктивный мутант) и «Нет».

полезная (непродуктивная мутантная) ссылка. Пожалуйста исправьте соответствует

просьба к автору чейнджлиста по улучшению теста

набор на основе заявленного мутанта; бесполезно соответствует

к ложной тревоге или вообще к обнаружению недействующего кода.

82% всех зарегистрированных мутантов с обратной связью были помечены

как продуктивный со стороны разработчиков. Обратите внимание, что это соотношение

агрегировать по всему набору данных. С момента инаугурации

Службы мутационного тестирования производительность выросла

с течением времени с 80% до 89%, поскольку мы обобщили

обратная связь о непродуктивных мутантах и созданном подавлении

правила экспертной функции , описанные в разделе 3. Это

означает, что более поздние мутации узлов, в которых были мутанты

окажется непродуктивным, будет подавлено, создавая

меньше непродуктивных мутантов с течением времени. Заявленные мутанты

без явной обратной связи с разработчиком не рассматриваются

анализ производительности.

**5.2 Подавление мутанта RQ1**

Чтобы сравнить наш подход к подавлению мутантов

с помощью традиционного мутагенеза мы (1) случайным образом отобрали образцы

5000 списков изменений из набора данных мутантов, (2) определено

сколько мутантов производит традиционный мутагенез, и

(3) сравнили результат с количеством созданных мутантов

по нашему подходу. (Поскольку традиционный анализ мутаций

непомерно дорого в масштабе, мы адаптировали нашу систему для

генерировать только все мутанты для выбранных списков изменений.) Рисунок

6 показаны результаты для трех стратегий: без подавления

(традиционный вариант), выберите по одному мутанту на линию и выберите одного мутанта

за строку после исключения засушливых узлов (наш подход). Мы

включить в анализ подход «1 на строку», чтобы оценить

индивидуальный вклад подавления засушливого узла,

помимо выборки одного мутанта на линию.

Как показано в Таблице 5, среднее количество сгенерированных

мутантов составляет 820 для традиционного мутагенеза, 77 для 1-линейного.

выбора и только 7 для выбора по одной строке. Следовательно,

наш подход к подавлению мутантов уменьшает количество

мутантов на два порядка. В таблице 5 также показаны

результаты U-теста Манна-Уитни, который подтверждает, что

распределения статистически значимо различаются.

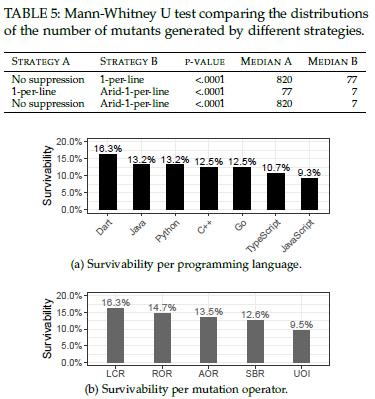


Рис. 7: Выживаемость мутантов.

Наш подход к подавлению мутантов генерирует менее

20 мутантов для большинства списков изменений; 25-й и 75-й процентили

3 и 19 соответственно. Напротив, 25-е

и 75-й процентиль для 1 на линию составляют 31 и 138 мутантов.

Традиционный мутагенез дает более 450 мутантов.

для большинства списков изменений (25-й и 75-й процентили составляют 460

и 1734 г. соответственно), еще раз подчеркнув, что такой подход

непрактично даже на уровне списка изменений. Представляя

сотни мутантов, большинство из которых не подлежат действию, чтобы

разработчик почти наверняка приведет к тому, что разработчик

полный отказ от мутационного тестирования.

**RQ1:** Arid-node suppression and 1-per-line selection significantly

reduce the number of mutants per changelist, with a

median of only 7 mutants per changelist (compared to 820

mutants for traditional mutagenesis).

**5.3 Выживаемость мутанта RQ2**

Выживаемость мутантов важна, потому что мы генерируем

максимум один мутант на линию — если этот мутант убит, нет

генерируется другой мутант. Чтобы действовать, мутанты должны

сообщать как можно скорее в процессе проверки кода,

как описано в разделе 4. Поэтому мы стремимся максимизировать

выживаемость мутантов, поскольку она напрямую влияет на численность

зарегистрированных мутантов.

В целом 87,5% всех созданных мутантов погибают

начальный набор тестов. Обратите внимание, что это не то же самое, что

традиционный показатель мутаций [30] (соотношение убитых мутантов

к общему числу мутантов), поскольку мутагенез

вероятностный и генерирует только подмножество всех мутантов. Этот

означает, что генерируется только часть всех возможных мутантов

и оценены, а многие другие мутанты никогда не создаются

потому что они связаны с засушливыми узлами.

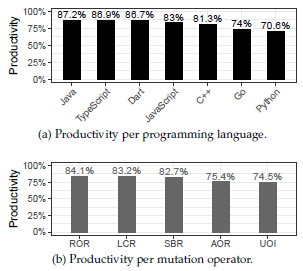


Рис. 8: Продуктивность мутантов.

В таблицах 3 и 4 показано распределение количества мутантов.

и выживаемость мутантов в разбивке по программированию

язык и оператор мутации. Рисунок 7 визуализирует

данные о выживаемости мутантов. Поскольку мутация SBR

оператор может быть применен практически к любому незасушливому узлу в

кода, неудивительно, что этот оператор мутации доминирует

количество мутантов, составляющих примерно 68% всех

мутанты. Хотя SBR является плодовитым и универсальным оператором мутации,

это также второй вариант с наименьшей вероятностью выжить в наборе тестов:

когда это применимо к списку изменений, сообщается о мутантах SBR.

при проверке кода с вероятностью 12,6%. Общий,

выживаемость мутантов одинакова для всех операторов мутации,

за заметным исключением УОИ, у которого есть живучесть

всего 9,5%. Выживаемость мутантов также одинакова

языки программирования, за исключением Dart, чей

выживаемость мутантов заметно выше. Мы предполагаем, что

это потому, что Dart в основном используется для веб-разработки.

у которого есть свои проблемы с тестированием.

**RQ2:** Different mutation operators result in different mutant

survivability; for example, the survival rate of LCR is almost

twice as high as that of UOI.

**5.4 Продуктивность мутанта RQ3**

Продуктивность мутантов является наиболее важным показателем, поскольку

он напрямую измеряет полезность зарегистрированного мутанта. С

мы создаем только одного мутанта в линии, этого мутанта

в идеале должен не только пережить набор тестов, но и быть

продуктивный, что позволяет разработчикам улучшать набор тестов

или сам исходный код. Учитывая высокую точность Google

и требования к действиям для выявления результатов кода

во время проверки кода мы полагаемся на отзывы разработчиков, поскольку

лучший доступный показатель продуктивности мутантов. Конкретно,

мы считаем мутантом разработчика с пометкой Пожалуйста, исправьте

быть более продуктивным, чем другие. Аналогично мы рассматриваем

мутант разработчик, отмеченный знаком «Бесполезно», чтобы быть менее продуктивным

чем другие. Сравниваем продуктивность мутантов

через операторы мутации и языки программирования.

На рисунке 8 показаны результаты, показывающие, что продуктивность мутантов

одинаково для всех операторов мутации, с AOR

а мутанты UOI заметно менее продуктивны. Для

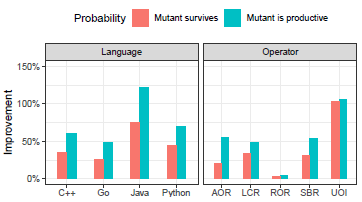


Рис. 9: Улучшения, достигнутые за счет выбора на основе контекста.

(улучшение 0% соответствует случайному выбору.)

Например, мутанты ROR продуктивны в 84,1% случаев,

тогда как мутанты UOI продуктивны только 74,5%

время. Различия между языками программирования заключаются

еще более выражено, поскольку мутанты Java являются продуктивными

87,2% случаев по сравнению с мутантами Python, которые

продуктивно 70,6% времени. Это может быть из-за кода

соглашения, общие сценарии использования языка, тестирование

фреймворки или просто отсутствие эвристики. Мы нашли

что код Python обычно требует большего количества тестов из-за

отсутствие компилятора. В отличие от Python, который в основном используется

для серверной части преимущественно используются JavaScript, TypeScript и Dart.

используемый в коде внешнего интерфейса, который радикально отличается.

**RQ3:** ROR, LCR, and SBR mutants show similar productivity,

whereas AOR and UOI mutants show noticeably lower productivity.

**5.5 Контекст мутации RQ4**

Мы исследуем, является ли контекстно-ориентированный отбор мутаций

операторы улучшают выживаемость и продуктивность мутантов.

В частности, мы определяем, является ли контекстно-ориентированный выбор

операторов мутации увеличивает вероятность сгенерированного

мутанта, чтобы выжить и привести к запросу «Пожалуйста, исправьте»,

по сравнению с базовым уровнем случайного выбора.

На рисунке 9 показано, что выбор операторов мутации на основе

в контексте AST узла, подвергающегося мутации, существенно

увеличивает вероятность выживания созданного мутанта

и в результате появится запрос «Исправьте». Хотя улучшения

различаться в зависимости от языка программирования и мутации

операторов, контекстно-ориентированный выбор неизменно превосходит

случайный выбор. Наибольшее повышение производительности

достигаются для UOI, AOR и SBR, которые генерируют

больше всего мутантов. Интуитивно эти улучшения означают

что выбор на основе контекста приводит к вдвое большему количеству продуктивных

UOI-мутанты (из всех сгенерированных мутантов), когда

по сравнению со случайным выбором. Рисунок 9 также показывает, к чему

В какой степени эти улучшения можно объяснить тем, что

просто выживает больше мутантов. Поскольку улучшения для

производительность возрастает даже больше, чем по живучести,

контекстно-ориентированный выбор не только приводит к увеличению количества сообщений

мутантов, но и в более высокой продуктивности этих мутантов.

В целом выживаемость увеличивается более чем на 40%, а

вероятность того, что рецензент попросит сгенерированного мутанта

быть фиксированным увеличивается почти на 50%.

Важно поместить эти улучшения в контекст.

Вероятностный анализ мутаций на основе различий агрессивно сокращает

сократилось число рассматриваемых мутантов с тысяч в

репрезентативный файл лишь для немногих и позволяет мутантам быть

эффективно представлены разработчикам в качестве потенциальных объектов тестирования.

Подход случайного выбора дает меньше выживших

мутанты меньшей продуктивности.

**RQ4:** Context-based selection improves the probability that a

generated mutant survives by more than 40% and the probability

that a generated mutant is productive by almost 50%.

**6 СВЯЗАННАЯ РАБОТА​**

Есть несколько направлений исследований, связанных с этим.

работа. Просто и др. предложил контекст программы на основе AST

модель прогнозирования эффективности мутантов [31]. Фернандес

и другие. разработал различные правила для Java-программ для обнаружения

эквивалентные и избыточные мутанты [32]. Первые результаты

перспективны для разработки стратегий отбора, которые превосходят

случайный выбор. Кроме того, Чжан и др. подержанная машина

научиться прогнозировать результаты мутаций, как на последовательных

версий данного проекта и между проектами [33]. Окончательно,

проект PIT делает тестирование мутаций полезным, практикуя

разработчиков и получил распространение в отрасли [16].

Большое внимание уделялось вычислительным затратам и

эквивалентная проблема мутанта [34]. Существует много внимания

на избежание избыточных мутантов, что приводит к увеличению

вычислительные затраты и увеличение количества мутаций [35],

и вместо этого отдавать предпочтение труднообнаружимым мутантам [36], [37] или

мутанты-доминаторы [38]. Графики включения мутантов имеют

аналогичные цели, но продуктивность мутантов гораздо более размыта

чем доминирование или подчинение.

Эффективность для мутантов в первую очередь определяется с точки зрения

избыточность и эквивалентность. Этот подход не учитывает

представление о том, что нередактантные мутанты могут быть непродуктивными

или что эквивалентные мутанты могут быть продуктивными [39].

По нашему опыту, сообщение об эквивалентных мутантах было

гораздо более простая проблема, чем сообщать о непродуктивных, неизбыточных

и неэквивалентные мутанты.

Наш подход к целевому отбору мутантов (раздел 4)

сравнивает контекст мутантов, используя хеши деревьев.

конкретная реализация была обусловлена необходимостью последовательности

и эффективность, чтобы можно было смотреть

создавать подобные контексты AST в реальном времени во время создания мутантов.

В частности, хеш-расстояния должны сохраняться в течение

время улучшить целевой отбор. Есть подходы

для программного обнаружения клонов [40], которые аналогичным образом используют древовидные расстояния

(например, [41], [42], [43], [44], [45]). Является ли альтернативой

измерения расстояний могут быть масштабированы для применения в

Google и смогут ли они еще больше улучшить целевую аудиторию

Выбор еще предстоит определить в дальнейшей работе.

Этот подход аналогичен древовидным подходам в

обнаружение клонов программного обеспечения [40], целью которого является обнаружение того, что

фрагмент кода — это копия некоторого исходного кода с или

без модификации. Методы на основе AST могут обнаружить

дополнительные категории модификаций, такие как имя идентификатора

изменения или вводы псевдонимов, которые не могут быть обнаружены на основе токенов,

и важна нечувствительность к именам переменных

для контекста мутации. Однако обнаружение клонов отличается

радикально в своей цели: он заботится об обнаружении кода с помощью

та же семантика, несмотря на внесенные синтаксические изменения

к этому. Хотя при обнаружении клонов может потребоваться обнаружить, что

алгоритм был скопирован, а затем немного изменен, например:

рекурсия, переписанная в эквивалентный итерационный алгоритм,

Контекст тестирования мутаций заботится только о соседних

Узлы AST: в итеративном алгоритме наиболее производительный

мутантами будут те, кто раньше процветал в таком коде, а не

те, которые преуспели в рекурсивном алгоритме. Чтобы

искать похожие контексты AST в реальном времени, как мутанты

создаются, нам нужен быстрый метод, сохраняющий хэш

расстояние во времени. За эту последовательность и эффективность

По причинам мы выбрали описанный подход хеширования дерева.

**7 ВЫВОДЫ**

Мутационное тестирование может эффективно управлять программным обеспечением.

тестирование и повышение качества программного обеспечения. Однако многие

мутанты представляют собой непродуктивные цели испытаний; написание тестов для

они не повышают эффективность набора тестов и, что еще хуже,

отрицательно влияет на ремонтопригодность тестов.

За последние шесть лет мы разработали масштабируемую

Подход к тестированию мутаций и правила подавления мутантов

что увеличило долю продуктивных мутантов, по оценкам

разработчиками. На ранних этапах проекта первоначальный

правила подавления мутантов улучшили соотношение продуктивных

мутанты от 15% до 80%. По мере развития продукта дополнительные

правила подавления мутантов улучшили продуктивность

до 89%. Три стратегии были ключом к успеху. Сначала мы

разработал стратегию поэтапного тестирования мутаций, сообщив

не более одного мутанта на строку кода — нацеленные на строки, которые

изменены и закрыты. Во-вторых, мы создали набор

основанных на правилах эвристик для подавления мутантов, основанных на

отзывы разработчиков и ручной анализ. В-третьих, мы разработали

вероятностный, целевой подход к отбору мутантов, который учитывает

контекст мутации и исторические результаты мутации.

Учитывая успех нашего подхода к тестированию мутаций и

положительные отзывы разработчиков, ожидаем, что и дальше

принятие командами разработчиков приведет к дополнительным

уточнения стратегий подавления и отбора. Более того,

Важным аспектом наших текущих исследований является

понять долгосрочные последствия мутационного тестирования на

поведение разработчика [20].