УДК 004.415.53

ПРИОРИТЕЗАЦИЯ ТЕСТОВ В РЕГРЕССИВНОМ ТЕСТИРОВАНИИ

А. Г. МАЛЫШЕВСКИЙ

Рассмотрена приоритезация (ранжирование тестов) в регрессивном тестировании для ускорения выявления ошибок в программном обеспечении при выполнении набора тестов. Приведены новые методы приоритезации для повышения эффективности и понижения затрат в случае их применения. Проведена оценка сравнительной эффективности как новых, так и известных из литературы методов.

ВВЕДЕНИЕ

Цель верификации программы — определить, удовлетворяет ли программа заданным требованиям и ее спецификации. Одним из видов верификации является тестирование (выполнение программы и проверка ее поведения на соответствие спецификациям). Обычно в тестировании используется набор тестов. Каждый тест состоит из множества входных значений (сценариев тестирования), применяемых к программе для проверки ее поведения. Назовем конечное множество тестов (или сценариев тестирования) T набором тестов. Одним из подходов тестирования может быть исчерпывающее тестирование, при котором набор тестов состоит из всех возможных допустимых входных значений программы. Однако в большинстве случаев такой подход не практичен. Поэтому набор создается, исходя из некоторого множества правил, определяющих его состав, называемого критерием адекватности тестов. Критерий адекватности выражает условия, которым должен удовлетворять набор тестов [1].

Регрессивное тестирование — это тестирование, применяемое после внесения изменений в программу для ее проверки. Оно используется в стадии сопровождения после исправления ошибок, адаптации программы к новой среде или увеличения ее производительности. Таким образом выполняется проверка того, что внесенные в программу модификации не только изменили программу в соответствии с новыми спецификациями и исправили найденные ошибки, но и не внесли новых ошибок [1]. Разработчики программного обеспечения часто сохраняют написанные ими тесты для их повторного использования в процессе эволюции программного обеспечения. Для проверки новой функциональности программы разработчики добавляют

© *А.Г. Малышевский,* 2006

новые тесты в набор, который в результате растет, а вместе с ним растет и стоимость регрессивного тестирования.

Так как регрессивное тестирование применяется многократно, его стоимость может существенно влиять на стоимость всего программного продукта. Поэтому даже небольшое уменьшение стоимости регрессивного тестирования может значительно снизить стоимость всего программного обеспечения. Уменьшая время тестирования, можно уменьшить время создания новой версии программы. Пример высокой стоимости регрессивного тестирования — программный продукт, тестирование которого занимает семь недель. Существует несколько подходов к уменьшению стоимости регрессивного тестирования: 1) в регрессивном отборе тестов (regression test selection) перед каждой сессией тестирования выбирается подмножество набора тестов, и затем оно применяется для проверки модифицированной программы; 2) в редукции набора тестов (test suite reduction), где в некоторый момент времени набор перманентно редуцируется, из данного набора удаляется часть тестов и не используется при будущих тестированиях.

В некоторых случаях, когда невозможно или не разрешается пропускать тесты в регрессивном тестировании, описанные два подхода неприемлемы, например, для программ, надежность которых является критичной (авионика или управление медицинским оборудованием). В данном случае для уменьшения стоимости регрессивного тестирования может быть применен иной подход: тесты упорядочиваются (приоритезируются) для регрессивного тестирования таким образом, чтобы более важные из них выполнялись перед менее важными. В методах приоритезации тесты сортируют таким образом, чтобы эффективнее достичь заданной цели [2, 3]. Примером такой цели может быть наиболее быстрое покрытие операторов программного кода, функций программы в порядке частоты их использования или подсистем в порядке частоты их сбоя в прошлом. Возможная цель приоритезации — увеличение скорости выявления ошибок набором тестов в процессе тестирования. Возросшая скорость выявления ошибок может обеспечить более раннюю обратную связь с регрессивно тестируемой системой и позволить разработчикам начать поиск местонахождения ошибок, а также их исправление раньше, чем это было бы возможно в ином случае. Такая обратная связь обеспечивает более ранние признаки того, что заданные цели еще не достигнуты и позволяет принимать более ранние стратегические решения о сроках релиза. Кроме этого, улучшенная скорость выявления ошибок повышает вероятность того, что в случае преждевременного прекращения процесса тестирования тесты, обеспечивающие наибольшую способность выявлять ошибки в сроки, выделенные под тестирование, уже были выполнены.

Рассмотрим результаты, известные из литературных источников.

Rothermel и другие авторы в работе [2] описывают эксперименты над несколькими небольшими программами на языке Си. Исследовано шесть методов приоритезации, включая приоритезацию тестов по общему и дополнительному покрытию операторов, ветвей, а также операторов, использующих потенциал выявления ошибок. Все рассмотренные в [2] методы показали некоторую способность улучшать значение метрики APFD по сравнению со случайным упорядочиванием. В целом методы, использую-

щие потенциал выявления ошибок, показали наилучшую эффективность. За ними следует приоритезация тестов по общему покрытию ветвей и операторов.

В работе [4] Rothermel и другие авторы расширили эти результаты, используя те же программы, что и в [2]. Здесь в роли ошибок выступали мутанты (измененные операторы). Подобно результатам предыдущих экспериментов приоритезация по дополнительному покрытию операторов, использующая потенциал выявления ошибок, достигла статистически значимых наилучших результатов.

Wong и другие авторы в работе [3] предлагают приоритезировать тесты в соответствии со следующим критерием: сортировать по увеличивающейся стоимости тестов на единицу дополнительного покрытия. Авторы ограничиваются приоритезацией подмножества тестов, выбранных безопасным методом регрессивного отбора, хотя оставшиеся тесты могут быть добавлены к концу списка для дальнейшего выполнения.

Jones и Harrold в работе [5] описывают метод приоритезации для наборов тестов с таким критерием адекватности, как покрытие решений модифицируемым условием (modified condition/decision coverage). Данный метод использует итеративный подход, обновляя информацию о тестах в процессе добавления их к отсортированной последовательности. Алгоритм создает список упорядоченных последовательностей тестов.

Srivastava и Thiagarajan в работе [6] показали метод приоритезации, основанный на покрытии базисных блоков (часть кода, имеющая только один вход и один выход). В данном алгоритме в каждой итерации выбирается тест, покрывающий наибольшее количество еще непокрытых модифицированных базисных блоков кода. Этот метод был применен к нескольким большим системам в Microsoft, демонстрируя возможность его эффективного использования. Приоритезированные тесты быстро достигают покрытия кода и могут рано выявлять ошибки. Однако в работе [6] нет сравнения предложенного авторами метода с другими известными методами и со случайным упорядочиванием тестов. Следовательно, невозможно сказать, увеличивает ли этот метод скорость выявления ошибок в сравнении с другими методами.

Kim и Porter в работе [7] демонстрируют метод, который они определяют как «приоритезация, основанная на прошлом». Здесь используется информация из предыдущих циклов регрессивного тестирования для лучшего информирования процесса выбора подмножества тестов из существующего набора в случае применения к модифицированной версии системы. Этот метод не является методом приоритезации с точки зрения формального определения, так как он не ранжирует тесты (основная характеристика определения приоритезации), а выбирает их подмножество, используя информацию о предыдущих тестированиях. Более точно он описывается как метод регрессивного отбора тестов [8]. Исходной информацией может быть результат выполнения теста, выявление ошибки или покрытие элемента программы (например, оператор, ветвь).

Avritzer и Weyuker в работе [9] описали методы генерирования тестов для программ, моделируемых цепями Маркова при наличии данных опера-

ционного профиля. Хотя авторы не используют термин «приоритезация», их методы генерируют тесты в таком порядке: бóльшие подмножества состояний программы, наиболее вероятные при ее нормальном использовании, покрываются как можно раньше (особенно при сортировке тестов в порядке увеличения вероятности того, что ошибки, которые могут привести к сбоям программы при ее нормальном использовании, выявятся раньше в процессе тестирования). Данный метод не рассматривает приоритезацию существующих тестов при тестировании модифицированной программы, однако он показывает пример использования приоритезации в случае недоступности набора тестов.

Несмотря на проведенные исследования в работах [2, 4, 5, 6, 7, 9], для успешной интеграции приоритезации в процесс регрессивного тестирования необходима разработка эффективных и дешевых методов, которые сделают ее применение целесообразным. В дополнение к проведенным ранее исследованиям на ограниченном наборе программ (объектов исследования), в данной работе проводятся эмпирические исследования на расширенном множестве подобных объектов, что позволит проверить возможность обобщения полученных ранее результатов и необходимость дальнейших исследований.

ОПРЕДЕЛЕНИЕ ПРИОРИТЕЗАЦИИ ТЕСТОВ

Дано: T — набор тестов; PT — множество перестановок T; f — функция $PT \to \mathbb{R}$.

Задача: найти такое $T' \in PT$, что для всех T'' из PT: $[f(T') \geq f(T'')]$, где PT — множество всех возможных перестановок T; f — функция, примененная к любой такой перестановке, вычисляет ценность этой перестановки.

МЕТОДЫ ПРИОРИТЕЗАЦИИ

Существуют различные методы достижения цели приоритезации. Например, для увеличения скорости выявления ошибок тесты могут быть приоритезированы в соответствии с выполнением модулей, имеющих тенденцию к сбоям в прошлом. Таким же образом можно приоритезировать тесты в соответствии с покрытием компонентов программного кода или охватом функциональностей, упомянутых в требованиях к программе. В любом случае задачей приоритезации является увеличение вероятности достижения заданной цели приоритезированным набором тестов по сравнению с неприоритезированным. В наших исследованиях цель приоритезации — увеличение скорости выявления ошибок в процессе выполнения набора тестов. Другими словами: так упорядочить тесты, чтобы как можно быстрее выявить ошибки.

Рассмотрим 14 методов приоритезации (табл. 1), разделенные на три группы.

Таблица 1. Методы приоритезации

Метка	Мнемоника	Метка	Мнемоника	Метка	Мнемоника
T1	random	T6	st-fep-addtl	T11	fn-fi-total
T2	optimal	T7	fn-total	T12	fn-fi-addtl
T3	st-total	T8	fn-addtl	T13	fn-fi-fep-total
T4	st-addtl	T9	fn-fep-total	T14	fn-fi-fep-addtl
T5	st-fep-total	T10	fn-fep-addtl		

Первая — контрольная. Содержит два метода: random и optimal.

Вторая — группа методов, использующих информацию на уровне операторов. Содержит четыре метода с мелкой гранулярностью (гранулярности уровня операторов).

Данные методы использовались в работе [2], здесь же мы их изучаем в контексте приоритезации, ориентированной на заданную версию программы (version-specific).

Третья — группа уровня функций. Содержит восемь методов крупной гранулярности (гранулярности на уровне функций), четыре из них похожи на методы уровня операторов, остальные четыре дополнительно используют информацию о вероятности существования ошибок. Опишем каждую из них отдельно.

Группа 1. Контрольные методы. Используются для оценки минимальной и максимальной эффективности приоритезации (в своем роде точек отсчета).

T1. Случайный порядок — случайное упорядочение тестов в наборе.

T2. Оптимальный порядок — метод оптимальной приоритезации (быстрейшего выявления ошибок). Имея информацию об ошибках и выявляющих их тестах, можно найти порядок тестов, при котором скорость выявления ошибок максимальна. Хотя в реальной ситуации такой подход не практичен, он обеспечивает верхнюю границу эффективности других эвристических методов.

Группа 2. Методы, использующие информацию на уровне операторов.

T3. Приоритезация по общему покрытию операторов. Вставляя в программу дополнительный код, можно определить для каждого теста, какие операторы им покрыты. Эти тесты можно приоритезировать в соответствии с количеством покрытых ими операторов, сортируя тесты в порядке убывания данного общего покрытия. Так как в общем случае невозможно получить данные о покрытии, избежав выполнения всех тестов, используется приближенная информация о покрытии операторов, полученная на основании предыдущего регрессивного тестирования, т.е. предыдущая версия программы.

T4. Приоритезация по дополнительному покрытию операторов. Приоритезация по общему покрытию операторов планирует тесты в порядке достижения совокупного покрытия операторов. Однако, выполнив тест и

покрыв некоторые операторы, целесообразно при последующем тестировании покрыть еще не покрытые операторы. Приоритезация по дополнительному покрытию операторов выбирает тест, достигший наибольшего покрытия еще не покрытых, повторяя этот процесс до тех пор, пока ни один из оставшихся тестов не сможет увеличить покрытие операторов. Когда это происходит, данный алгоритм применяется рекурсивно к оставшимся тестам.

T5. Приоритезация по общему покрытию операторов с учетом FEP. Выявление тестом ошибки зависит не только от выполнения им компонента кода, содержащего ошибку, но также и от вероятности сбоя в результате этой ошибки [10]. Необходимо выяснить, можно ли использовать эту вероятность для увеличения эффективности приоритезации с точки зрения скорости выявления ошибок. Для оценки потенциала выявления ошибок FEP (fault-exposing-potential) теста был использован мутационный анализ [11].

Дана программа P и набор тестов T. Для каждого теста t из T и для каждого оператора s в P найдена оценка мутаций (mutation score) $ms(s,t)$ теста t для оператора s (отношение количества мутантов оператора s, убитых (выявленных) тестом t, к общему количеству мутантов оператора s). Затем для каждого теста $t \in T$ вычислялась ценность путем суммирования всех значений $ms(s,t)$. Приоритезация по общему покрытию операторов с учетом FEP сортирует тесты в порядке убывания полученных значений ценности. Из-за высокой стоимости получения значений FEP данный метод приоритезации дороже методов, базирующихся только на покрытии операторов. Однако, если данный метод будет достаточно эффективным, то станет целесообразным поиск приемлемых по стоимости методов аппроксимации значений FEP.

T6. Приоритезация по дополнительному покрытию операторов с учетом FEP. Аналогично трансформации приоритезации по общему покрытию операторов в приоритезацию по дополнительному покрытию трансформируем предыдущий метод с учетом FEP. В новом методе после планирования теста *t* понизим вес всех других тестов, покрывающих операторы, покрытые данным тестом. Затем данный процесс повторяется до полного упорядочивания тестов.

Группа 3. Методы, работающие на уровне функций.

T7. Приоритезация по общему покрытию функций аналогична приоритезации по общему покрытию операторов, но оперирует на уровне функций.

T8. Приоритезация по дополнительному покрытию функций аналогична приоритезации по дополнительному покрытию операторов, но оперирует на уровне функций.

T9. Приоритезация по общему покрытию функций с учетом FEP аналогична приоритезации по общему покрытию операторов с учетом FEP, но оперирует на уровне функций.

T10. Приоритезация по дополнительному покрытию функций с учетом FEP аналогична приоритезации по дополнительному покрытию операторов с учетом FEP, но оперирует на уровне функций.

T11. Приоритезация по общему покрытию функций с использованием FI. Присутствие ошибок не является равновероятным в каждой функции. Уровень склонности к содержанию ошибок (fault proneness) может быть ассоциирован с измеряемыми атрибутами программного кода [12, 13]. Для того чтобы отразить уровень склонности к ошибкам, используется индекс ошибок FI (fault index), базирующийся на методе главных компонент [14]. Определение значений FI для каждой функции требует вычисления значений FI для базовой и для новой версий, а также их сравнения. Таким образом, каждой функции присваивается абсолютное значение FI, показывающее ее склонность к содержанию ошибок с учетом сложности модификаций этой функции. При полученных значениях FI приоритезация по общему покрытию FI выполняется подобно приоритезации по общему покрытию функций. Для каждого теста просуммируем значения FI каждой функции, покрытой этим тестом. Затем упорядочим тесты в порядке убывания этих сумм. Таким образом тесты планируются согласно покрытию ими склонных к ошибкам функциям.

T12. Приоритезация по дополнительному покрытию FI подобна приоритезации по дополнительному покрытию функций, за исключением того, что ценность каждого теста вычисляется как сумма значений FI дополнительно покрытых функций.

T13. Приоритезация по общему покрытию функций с применением FI и FEP. Используя приближенную оценку потенциала выявления ошибок и оценку уровня склонности к ошибкам, что дает большую скорость выявления ошибок, мы применили приоритезацию по общему покрытию функций с учетом FI. В случаях, когда два и более теста имеют равные значения ценности, применили приоритезацию по общему покрытию функций с учетом FEP.

T14. Приоритезация по дополнительному покрытию функций с применением FI и FEP. Чтобы свести метод T13 к данному, используем приоритезацию по дополнительному покрытию функций с учетом FI. В случаях, когда два и более теста имеют равные значения ценности, сортируем их по общему покрытию функций с учетом FEP.

ЭМПИРИЧЕСКИЕ ИССЛЕДОВАНИЯ

Для изучения возможности применения приоритезации на практике проведено несколько эмпирических исследований с целью получения ответов на такие вопросы: способна ли приоритезация, ориентированная на заданную версию программы, увеличить скорость выявления ошибок и как сравниваются методы приоритезации низкой гранулярности (уровень операторов) с методами крупной гранулярности (уровень функций) с точки зрения скорости выявления ошибок, а также может ли использование оценки склонности к ошибкам повысить эффективность методов приоритезации.

Эффективность приоритезации и ее измерение

Для анализа эффективности приоритезации необходимо оценить ее количественно. Введем метрику, оценивающую скорость выявления ошибок набо-

ром тестов. Назовем ее «взвешенное среднее процента выявленных ошибок» APFD (weighted average of the percentage of faults detected). Метрика APFD измеряет скорость выявления ошибок в интервале (0...100). Чем больше значение метрики, тем быстрее выявляются ошибки.

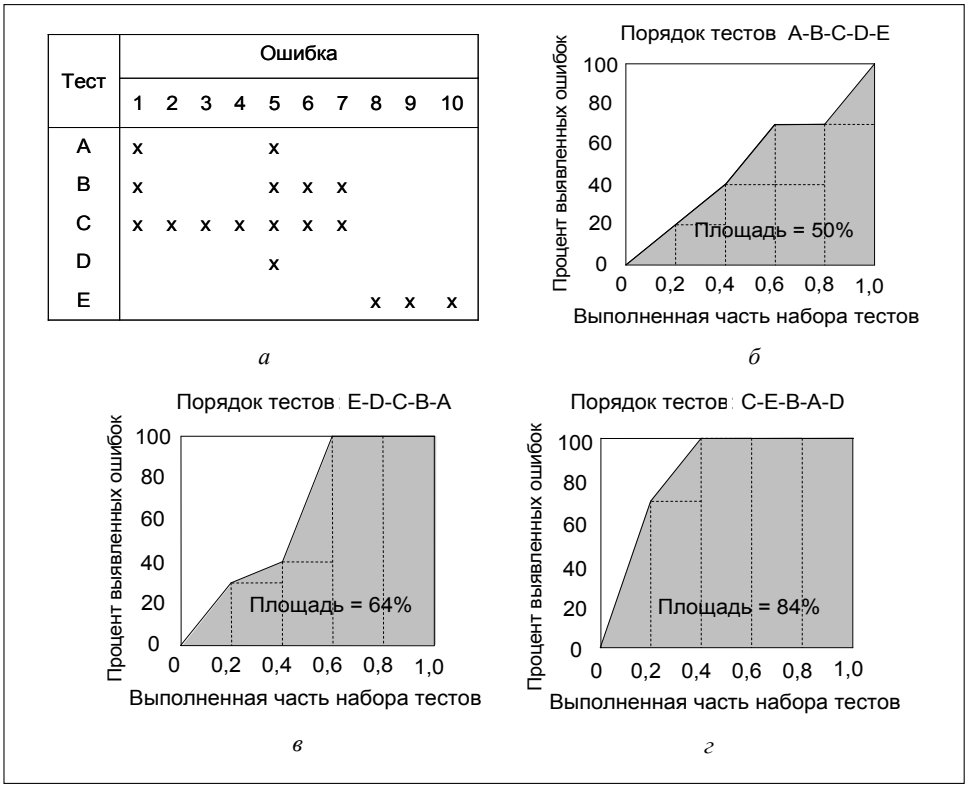

Рис. 1. Пример метрики APDF

Для иллюстрации данной метрики рассмотрим программу с десятью ошибками и набором из пяти тестов (A, B, C, D, E). Рис. 1, *а* демонстрирует способность данных тестов выявлять эти ошибки. Допустим, что мы разместим тесты в порядке A-B-C-D-E, формируя таким образом приоритезированный набор тестов T1. Рис. 1, *б* показывает процент выявленных ошибок относительно уже выполненной части набора тестов T1. После выполнения теста A выявлены две ошибки из десяти. Таким образом 20% ошибок выявлены после выполнения 20% набора тестов T1. После выполнения теста B две другие ошибки выявлены, и таким образом 40% ошибок выявлены после использования 40% набора тестов. На рис. 1, *б* пространство в прямоугольниках с прерывистыми линиями показывает взвешенное среднее процента выявленных ошибок в течение выполнения соответствующей части набора тестов. Сплошные линии, связывающие углы прямоугольников, интерполируют прирост процента выявленных ошибок. Эта интерполяция — поправка гранулярности (когда только небольшое количество тестов составляет их набор). Чем больше набор тестов, тем меньше эта поправка. Таким образом, площадь под кривой будет метрикой APFD приоритезированного набора тестов. В данном примере APFD равна 50%. Рис. 1, *в* отражает ситуацию,

когда порядок тестов изменился на E-D-C-B-A, получив набор тестов с более быстрым выявлением ошибок, чем набор T1 с соответствующим APFD, равным 64%. На рис. 1, *г* показано влияние использования приоритезированного набора тестов T3 в порядке C-E-B-A-D. Данный порядок (в этом примере оптимальный) раньше других выявляет большинство ошибок (APFD равен 84%).

Метрика APFD вычисляется по формуле $APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n}$ и выражается в процентах, где n — количество тестов в наборе; m — количество ошибок, выявленных данным набором тестов; TF_i — порядковый номер теста (начинающийся с единицы) в упорядоченном наборе, первым выявившим i-ю ошибку.

Инструментарий эмпирических исследований

В роли объектов исследований использованы восемь программ на языке Си. Семь из них в виде наборов тестов и версий с внесенными ошибками подготовлены исследователями из Siemens Corporate Research для изучения способности к выявлению ошибок при использовании критериев покрытия по потокам управления (control-flow) и по потокам данных (data-flow). Мы назовем их «Siemens-программы». Восьмая программа, *space*, разработана для ЕКА.

Siemens-программы решают следующие задачи: *tcas* предупреждает столкновение самолетов, *schedule2* и *schedule* планирует с использованием приоритетов, *tot_info* вычисляет некоторую статистику, *print_tokens* и *print_tokens2* — лексические анализаторы и *replace* осуществляет поиск по шаблону и замену. Используя прием разделения категорий, для каждой программы исследователи в Siemens создали *пул тестов*, состоящий из функциональных тестов. Затем они вручную добавили к данному пулу разработанные структурные тесты, чтобы гарантировать покрытие как минимум 30 тестами каждого оператора, ребра и *du*-пары (если это возможно теоретически) в базовой версии программы или ее графа потоков управления. Для каждой программы путем модификации кода в базовой версии исследователи создали версии, содержащие ошибки. В большинстве случаев они изменили единственную строку кода, и только в нескольких случаях модифицировали от двух до пяти строк кода. Целью было внедрение максимально реалистичных ошибок. Исследователи сохранили только ошибки, выявляемые, как минимум, тремя и, как максимум, 350-ю тестами в соответствующем пуле тестов.

Программа *space* — это интерпретатор языка для задания конфигурации массива ADL. Эта программа содержит 35 версий, каждая из которых имеет единственную ошибку. Все эти ошибки реальные и были обнаружены либо в ходе разработки программы, либо в процессе использования ее в исследованиях. Для данной программы создан тестовый пул за две стадии. Изначально пул состоял из 10 000 случайным образом сгенерированных тес-

тов, созданных Vokolos и Frankl. Затем новые тесты добавлялись до тех пор, пока каждое исполняемое ребро в графе потоков управления программой не было покрыто как минимум 30 тестами. В результате получен пул с 13585 тестами. При создании наборов тестов для данных программ использовались пулы тестов для базовых версий и информация о покрытии элементов программы тестами в этих пулах с целью генерирования 1000 адекватных покрытию ветвлений наборов тестов для каждой программы. В наших исследованиях для каждой программы случайным образом выбрали 50. Требовались программы с разным количеством ошибок. Каждая программа изначально обеспечивалась корректной базовой версией и несколькими версиями, содержащими по одной ошибке (версии 1-го порядка). Среди этих версий идентифицированы все версии, не мешающие друг другу, другими словами, все ошибки, которые могут быть внесены в базовую версию и одновременно сосуществовать. Затем были созданы версии более высокого порядка путем комбинирования не мешающих друг другу версий 1-го порядка. Чтобы увеличить достоверность наших экспериментов, для каждой программы было получено одинаковое количество версий (29) со случайным числом ошибок в каждой.

Инструментарий для приоритезации и анализа. С целью получения информации о покрытии операторов и функций тестами, о графах потоков управления и об оценках мутаций использовались те же инструменты, что и в работах [2, 4, 15]. Были реализованы методы приоритезации (табл. 1). Чтобы получить информацию о склонности функций содержать ошибки, использовались три инструмента [16, 17]: измеряющий исходный программный код для вычисления метрики сложности кода, сам генератор значения склонности кода к ошибкам и сравнивающий инструмент для оценки каждой версии относительно базовой.

Далее детально опишем эмпирические исследования и их результаты. На рис. 2 показана диаграмма размаха объединенных данных всех программ («all program total») и отдельные диаграммы для каждой из них, демонстрируюшие распределение значений APFD для 14 методов приоритезации (табл. 1).

Приоритезация, ориентированная на заданную версию. Первое исследование рассматривает, может ли приоритезация, ориентированная на заданную версию, улучшить способность ускорять выявление ошибок наборами тестов. Предполагаем, что различия в гранулярности могут вызвать значительные различия в выявлении ошибок, поэтому нами проведен эксперимент, состоящий из двух частей: первая — методы на уровне операторов и вторая — методы на уровне функций. Обе части выполнены в одинаковом факториальном дизайне (исследованы все комбинации на всех уровнях всех факторов). Факторами являлись программа и метод приоритезации. В каждой программе было восемь уровней с 29 версиями и 50 наборами тестов. Внутри методов — четыре уровня. В первой части рассматривались методы st-total, st-addtl, st-fep-total и st-fep-addtl, а во второй — fn-total, fn-addtl, fn-fep-total и fn-fep-addtl. Методы optimal и random [2] статистически значимо отличались от остальных методов, поэтому в данных экспериментах они

исключены из статистического анализа (представлены здесь лишь как точки отсчета).

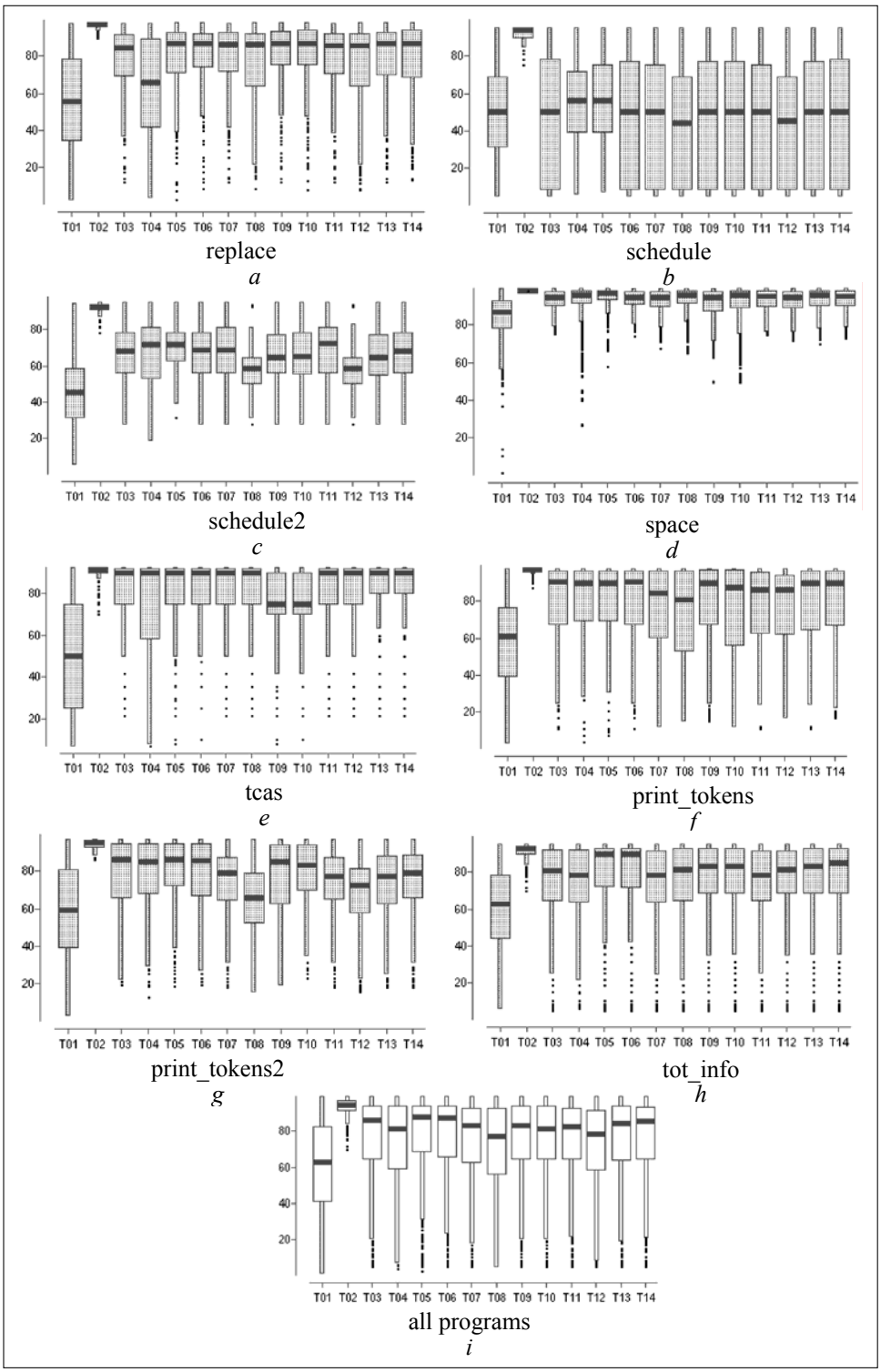

Рис. 2. Диаграммы размаха значений APFD для всех программ (горизонтальные оси — методы (табл. 1), вертикальные — значения APFD)

Проведен дисперсионный анализ (ANOVA) для изучения основного влияния факторов и взаимодействия между ними. В верхней половине табл. 2 приведены результаты первой части эксперимента при рассмотрении всех программ. Результаты показывают, что имеется достаточно статистических оснований для отклонения нулевой гипотезы: средние значения APFD, полученные различными методами, на уровне операторов отличаются. Однако из анализа видно, что существует значительная взаимосвязь между методами и программами. Различия между методами не идентичны для каждой из программ. Таким образом, их необходимо осторожно интерпретировать в индивидуальном порядке. Средние значения APFD ранжировали методы, дисперсионный анализ оценил, различались ли методы, а процедура множественного сравнения, использующая критерий Бонферрони, численно оценила различия методов. В нижней половине табл. 2 приведены результаты для методов на уровне операторов. Методы с одинаковой буквой группирования не имеют статистически значимых различий с точки зрения метрики APFD. Например, в st-fep-total среднее значение больше, чем в st-total, но они сгруппированы вместе, так как между ними нет статистически значимых различий. С другой стороны, st-fep-addtl — метод, использующий FEP-информацию лучше, чем другие методы, и данные различия являются статистически значимыми.

Таблица 2. Дисперсионный анализ и критерий Бонферрони, методы уровня операторов, все программы

Источник	Количество степеней свободы	Средний квадрат	F	P > F
Эффект	31	146227,87	397,72	0,0001
Ошибка	31836	367,66		
Минимальное значимое различие 0,80				
Группа	Среднее значение	Метод		
A	78,88	st-fep-addtl		
B	76,99	st-fep-total		
B	76,30	st-total		
C	74,44	st-addtl		

В табл. 3 показаны аналогичные результаты дисперсионного анализа и критерия Бонферрони для второй части эксперимента. Эффекты взаимодействия между методами и программами являлись значимыми для данных методов на уровне функций. Результаты также показывают статистически значимые различия между методами. Методы заняли порядок, идентичный их двойникам на уровне операторов: первый — fn-fep-addtl, второй — fn-fep-total, третий — fn-total и последний — fn-addtl. Однако в данном случае три наилучших метода не имели статистически значимого различия между собой. Как минимум, данный результат наводит на мысль о том, что метод, применявшийся для приближенной оценки значений FEP на уровне функций может не быть таким же эффективным, как метод для приближенной

оценки этих значений на уровне операторов. Чтобы определить, обобщается ли данный результат и можно ли найти более эффективные методы оценки FEP на уровне функций, необходимы дальнейшие исследования.

Таблица. 3. Дисперсионный анализ и критерий Бонферрони, методы уровня функций, все программы

Источник	Количество степеней свободы	Средний квадрат	F	P > F
Эффект	31	169080,15	436,93	0,0001
Ошибка	31836	386,96		
Минимальное значимое различие 0,82				
Группа	Среднее значение			Метод
A	75,59			fn-fep-addtl
A	75,48			fn-fep-total
A	75,09			fn-total
B	71,66			fn-addtl

Влияние гранулярности. Во втором исследовании сравниваются методы приоритезации с мелкой и крупной гранулярностью. Анализ данных показал, что гранулярность имеет влияние на значения APFD. Это очевидно, если посмотреть на диаграммы размаха, где для всех случаев средние значения APFD для методов на уровне функций были меньше, чем для соответствующих методов на уровне операторов. Например, среднее значение APFD для fn-fep-addtl равно 75,59, но для st-fep-addtl — 78,88. Каждый метод на уровне функций имеет меньшее значение APFD, чем каждый метод на уровне операторов. Выполнен эксперимент (эксперимент 2), подобный описанным выше. Проведены сравнения между следующими парами методов: (st-total, fn-total), (st-addtl, fn-addtl), (st-fep-total, fn-fep-total) и (st-fep-addtl, fn-fep-addtl). Для данных четырех пар методов различные уровни гранулярности имели основное влияние на значение APFD. Таким образом, несмотря на различное ранжирование в обеих частях первого эксперимента, статистической значимости было достаточно для подтверждения более высокой эффективности методов на уровне операторов по сравнению с методами на уровне функций.

Использование информации о склонности кода к ошибкам — третье исследование, направленное на увеличение скорости выявления ошибок с учетом оценки склонности кода к ошибкам. Предполагалось, что использование такой информации в методах приоритезации увеличило бы их эффективность. Разработан эксперимент (эксперимент 3) для изучения данной гипотезы на уровне функций. Он подобен второй части первого исследования за исключением добавления четырех новых методов: fn-fi-total, fn-fi-addtl, fn-fi-fep-total и fn-fi-fep-addtl. Дисперсионный анализ полученных в этом эксперименте данных (табл. 4) показал, что имела место статистическая значимость различия в эффективности этих методов. Результаты критерия Бонферрони оказались неожиданными. Несмотря на то, что метод fn-fi-fep-addtl имел наибольшее среднее значение APFD, он не отличался ста-

тистически значимо от fn-fep-addtl. Это означает что комбинация метрик FEP и FI (склонности к ошибкам) не увеличила эффективности методов с точки зрения метрики APFD. Отсутствие значимых различий имело также место и в других случаях, где использовалась оценка склонности к ошибкам, что произошло с такими парами соответствующих методов, как fn-fi-fep-total и fn-fep-total, fn-fi-total и fn-total, а также fn-fi-addtl и fn-addtl.

Таблица 4. Дисперсионный анализ и критерий Бонферрони, все методы приоритезации на уровне функций, все программы

Источник	Количество степеней свободы	Средний квадрат	F	P > F
Эффект	63	166395,24	440,71	0,0001
Ошибка	63736	377,59		
Минимальное значимое различие 0,96				
Группа	Среднее значение		Метод	
A	76,34		fn-fi-fep-addtl	
A B	75,92		fn-fi-fep-total	
A B	75,63		fn-fi-total	
A B	75,59		fn-fep-addtl	
A B	75,49		fn-fep-total	
B	75,09		fn-total	
C	72,62		fn-fi-addtl	
C	71,66		fn-addtl	

Итак, возможно, примененные методы оценивания FI не улучшили эффективность предложенных методов приоритезации. Такие результаты противоречат как нашим ожиданиям, так и результатам ранее опубликованных исследований [16]. Распределение ошибок в программах могло быть одной из возможных причин этого: иногда версии программ содержали только по единственной ошибке в программном коде. В таких случаях значение метрики FI сводилось к простейшему критерию: изменилась ли данная функция, снизив способность быть индикатором склонности к ошибкам. Хотя эта гипотеза требует дальнейших исследований, можно предположить, что относительная польза методов приоритезации для регрессивного тестирования зависит от свойств тестируемой программы, поэтому необходимо прогнозировать эффективность того или иного метода приоритезации в каждой конкретной ситуации.

Сравнительный анализ. Для построения общей картины всех рассмотренных методов приоритезации использовались дисперсионный анализ и критерий Бонферрони для всех методов, включая optimal и random (табл. 5). Как и ожидалось, дисперсионный анализ показал значительные различия между методами, а критерий Бонферрони сгруппировал их, подтверждая наши исследования. Стало очевидно: метод *optimal* значительно лучше любого другого метода. Это указывает на то, что методы приоритеза-

ции могут быть улучшены. Однако все методы превзошли случайное упорядочивание. Другое наблюдение: некоторые методы на уровне функций более эффективны по сравнению с методами на уровне операторов.

Таблица 5. Дисперсионный анализ и критерий Бонферрони, все методы, все программы

Источник	Количество степеней свободы	Средний квадрат	F	P > F
Эффект	111	190818,46	529,84	0,0001
Ошибка	111426	360,14		
Минимальное значимое различие 1,039				

Группа		Среднее значение	Метод
A		94,24	optimal
B		78,88	st-fep-addtl
	C	76,99	st-fep-total
D	C	76,34	fn-fi-fep-addtl
D	C	76,30	st-total
D	E	75,92	fn-fi-fep-total
D	E	75,63	fn-fi-total
D	E	75,59	fn-fep-addtl
D	E	75,49	fn-fep-total
F	E	75,09	fn-total
F		74,44	st-addtl
G		72,62	fn-fi-addtl
G		71,66	fn-addtl
H		59,73	random

ВЫВОДЫ

Результаты показывают статистически значимые различия в эффективности методов приоритезации с точки зрения влияния на скорость выявления ошибок. Конечно, отличия в скорости выявления ошибок могут и не иметь практического значения. Например, если время выполнения всех тестов является небольшим, данные отклонения могут быть незначительными. Однако, когда такое время достаточно большое, разница может стать весьма значительной. В данной работе эмпирически исследованы возможности увеличения скорости выявления ошибок наборами тестов нескольких методов приоритезации. Наши исследования ориентированы на приоритезацию заданной версии программы, для которой ранжируются тесты и измеряется скорость выявления ошибок. Описанные исследования имеют несколько практических результатов. Методы могут увеличить скорость выявления ошибок. Важен тот факт, что методы на уровне функций могут быть эффективными, кроме того, они дешевле и для них проще получить информацию. Однако методы на уровне операторов могут обеспечить более высокий уро-

вень эффективности, и потому предпочтительней в случае большой стоимости задержек в выявлении ошибок. С другой стороны, наши исследования о внедрении в процесс приоритезации информации о склонности к ошибкам показали, что, вопреки ожиданиям, внедрение этой информации существенно не улучшило эффективности приоритезации, следовательно, еще рано применять такой подход на практике.

Результаты подсказывают несколько направлений будущих исследований. В первую очередь, необходимо обобщить полученные результаты. Различия в эффективности рассмотренных методов приоритезации требуют дальнейших исследований факторов, влияющих на их эффективность. Было бы полезно разработать методы прогнозирования, которые для заданной программы, набора тестов и модификаций могли бы выбрать самый эффективный метод приоритезации. Необходимо изучить альтернативные цели приоритезации и альтернативные измерения ее эффективности. И, наконец, необходим поиск более эффективных методов приоритезации и методов аппроксимации оценок FEP и FI. Цель данной работы — создать для разработчиков программного обеспечения методы улучшения процессов регрессивного тестирования.

Автор благодарит Г. Ротермела и С. Элбаума за участие в проведении описанных исследований.

ЛИТЕРАТУРА

1. *Ghezzi C., Jazayeri M., Mandrioli D.* Fundamentals of Software Engineering. — Upper Saddle River: Prentice Hall, 1991. — 573 p.
2. *Rothermel G., Untch R., Chu C., Harrold M.J.* Test case prioritization: an empirical study // In Proceedings of the International Conference on Software Maintenance. — 1999. — P. 179–188.
3. *Wong W., Horgan J., London S., Agrawal H.* A study of effective regression testing in practice // In Proceedings of the Eighth International Symposium on Software Reliability Engineering. — 1997. — P. 230–238.
4. *Rothermel G., Untch R.H., Chu C., Harrold M.J.* Test case prioritization // IEEE Transactions on Software Engineering. — 2001. — **27**, № 10. — P. 929–948.
5. *Jones J.A., Harrold M.J.* Test-suite reduction and prioritization for modified condition/decision coverage // In Proceedings of the International Conference on Software Maintenance. — 2001. — P. 92–101.
6. *Srivastava A., Thiagarajan J.* Effectively prioritizing tests in development environment // In Proceedings of the International Symposium on Software Testing and Analysis. — 2002. — P. 97–106.
7. *Kim J.-M., Porter A.* A history-based test prioritization technique for regression testing in resource constrained environments // In Proceedings of the International Conference on Software Engineering. — 2002. — P. 119–129.
8. *Rothermel G., Harrold M.J.* Analyzing regression test selection techniques // IEEE Transactions on Software Engineering. — 1996. — **22**, № 8. — P. 529–551.
9. *Avritzer A., Weyuker E.J.* The automatic generation of load test suites and the assessment of the resulting software // IEEE Transactions on Software Engineering. — 1995. — **21**, № 9. — P. 705–716.

10. *Voas J.* PIE: A dynamic failure-based technique // IEEE Transactions on Software Engineering. — 1992. — P. 717–727.
11. *DeMillo R.A., Lipton R.J., Sayward F.G.* Hints on Test Data Selection: Help for the Practicing Programmer // Computer. — 1978. — **11**, № 4. — P. 34–41.
12. *Khoshgoftaar T.M., Munson J.C.* Predicting software development errors using complexity metrics // Journal on Selected Areas in Communications. — 1990. — **8**, № 2. — P. 253–261.
13. *Munson J.C.* Software measurement: Problems and practice // Annals of Software Engineering. — 1995. — **1**, № 1. — P. 255–285.
14. *Elbaum S.G., Munson J.C.* Code churn: A measure for estimating the impact of code change // In Proceedings of the International Conference on Software Maintenance. — 1998. — P. 24–31.
15. *Elbaum S., Malishevsky A., Rothermel G.* Prioritizing test cases for regression testing // In Proceedings of the International Symposium on Software Testing and Analysis. — 2000. — P. 102–112.
16. *Munson J.C., Elbaum S.G., Karcich R.M., Wilcox J.P.* Software risk assessment through software measurement and modeling // In Proceedings of the IEEE Aerospace Conference. — 1998. — P. 137–147.
17. *Elbaum S.G., Munson J.C.* Software evolution and the code fault introduction process // Empirical Software Engineering Journal. — 1999. — **4,** № 3. — P. 241–262.

Поступила 04.04.2006