Об одном методе сокращения набора тестов

Д.Ю. Кичигин

Аннотация. В статье излагается метод сокращения набора тестов для регрессионного тестирования, заключающийся в построении модели поведения программы на наборе тестов и последующей фильтрации тестового набора с помощью построенной модели. Модель поведения программы строится в терминах последовательностей системных вызовов, совершенных программой во время своего выполнения.

1. Введение

Темпы развития современного программного обеспечения и особенно увеличение количества вносимых в него модификаций приводят к увеличению затрат на проведение регрессионного тестирования. В связи с этим достаточно актуальной является проблема уменьшения стоимости регрессионного тестирования. Сокращение набора тестов [1,10,13] является одним из способов уменьшения затрат на проведение регрессионного тестирования.

Целью задачи сокращения набора тестов является создание набора тестов меньшего размера, но при этом, желательно, с таким же уровнем обнаружения ошибок [1]. Такое сокращение позволяет уменьшить затраты на сопровождение тестового набора и на выполнение на нем тестируемого программного обеспечения [1], что, в свою очередь, позволяет сократить затраты на проведение регрессионного тестирования в целом.

Традиционный подход к решению задачи сокращения набора тестов заключается в «отсеивании» тестов из первоначального набора таким образом, чтобы сохранялся уровень его адекватности в терминах некоторого выбранного критерия адекватности, и основывается на использовании статического анализа и/или инструментирования исходного текста программного обеспечения (ПО). При таком подходе решение задачи, как правило, состоит из трех основных этапов. На первом этапе происходит инструментирование кода тестируемого ПО и запуск ПО на наборе тестов. На втором этапе собирается информация о достигнутом покрытии кода. Наконец, на третьем этапе происходит сокращение набора тестов на основе собранной информации о покрытии [1].

Современные тенденции разработки программного обеспечения ставят новые условия для методов сокращения набора тестов и во многом ограничивают область применения существующих методов. Во-первых, растет популярность использования готовых программных компонентов. Такие компоненты, как правило, поставляются без исходного текста, что ограничивает применимость методов, основанных на анализе или инструментировании исходного кода программы [1]. Во-вторых, программное обеспечение зачастую разрабатывается с использованием разных языков программирования, что ограничивает доступность средств анализа и инструментирования исходного кода, так как многие из них ориентированы на работу с конкретным языком программирования [1]. В третьих, возрастающая сложность современного программного обеспечения, особенно увеличение его модульности, делает неудобным и повышает затраты на использование средств инструментирования кода, так как в этом случае увеличивается количество инструментируемых модулей и, соответственно, возрастает сложность анализа покрытия.

Рассмотренные обстоятельства делают актуальной разработку новых методов, способных работать в указанных условиях. В данной работе предлагается новый метод сокращения набора тестов, основанный на анализе поведения программы на наборе тестов и использующий для этого короткие последовательности системных вызовов. Предлагаемый метод не зависит от языка программирования, на котором производилась разработка ПО, не требует доступа к исходному коду и не предполагает инструментирования кода ПО, что существенно расширяет область его применения и облегчает использование.

Статья организована следующим образом. В разделе 2 рассмотрены основные существующие методы сокращения набора тестов. Раздел 3 посвящен описанию предлагаемого метода сокращения набора тестов. В разделе 4 приводится описание проведенного эксперимента и его результатов. В конце работы делаются выводы и рассматриваются направления будущих исследований.

2. Существующие решения

Существующие методы сокращения набора тестов [1,10,11,12,13] объединяет общий подход, который состоит в построении набора тестов меньшего размера, но при этом эквивалентного первоначальному набору в терминах выбранного покрытия кода программы. В результате полученный сокращенный набор тестов замещает собой первоначальный и используется для проведения регрессионного тестирования.

Существующие методы сокращения набора тестов в основном различаются видом используемого покрытия. В работе [13] для сокращения набора тестов используется покрытие «всех использований» (*«all-uses» coverage*). Это покрытие является частным случаем покрытия ассоциаций вида определение-

использование (*definitions-use associations coverage*) основанного на модели потока данных в программе [15]. Анализ этого покрытия основывается на изучении вхождений переменных в программу и рассмотрении путей выполнения программы от места определения переменной (*variable definition*) до места ее использования (*variable use*). В [10] рассматривается метод сокращения наборов на основе покрытия ребер[1] гипотетического графа потока управления программы. Для анализа покрытия ребер используется исходный код программы [14]. В работе [12] для сокращения набора тестов используется покрытие MC/DC (*Modified Condition/Decision Coverage*). Покрытие MC/DC определяется в терминах языка программирования, используемого при написании программы, и является развитием покрытия условий (*conditions coverage*) [16]. В [11] предлагается метод сокращения набора тестов, основанный на комбинации покрытия блоков программы и ассоциированных с выполнением теста ресурсозатрат. Блоком называется последовательность выражений программы, не содержащая ветвлений; таким образом, покрытие блоков эквивалентно покрытию выражений.

Общей особенностью перечисленных подходов является то, что для их применения необходим доступ и инструментирование исходного либо объектного кода программы, что подвергает эти методы указанным во введении ограничениям. Немного особняком стоит работа [1], где авторы в качестве элементов покрытия используют стек вызовов функций программы (*call stack*) и указывают, что такой подход может применяться и без доступа к исходному коду программы. Однако в эксперименте, проведенном авторами этой работы, для мониторинга стека вызовов функций требуется знать сигнатуры внутренних функций программы, для получения которых используется доступ к исходному тексту программы. Поэтому, по крайней мере, пока также нельзя говорить, что данная методика применима в условиях указанных выше ограничений.

3. Описание метода

В этой работе рассматривается тестирование программного обеспечения, работающего под управлением операционной системы семейства UNIX. Предлагаемый метод основывается на построении моделей поведения программы на наборе тестов и последующем использовании их для сокращения набора тестов. Для построения модели поведения программы используются последовательности системных вызовов, выполненные программой во время своей работы. Для мониторинга выполнения системных вызовов используется команда *ktrace*, входящая в состав операционной

[1] Критерий покрытия ребер похож на критерий покрытия ветвлений (*decision coverage*), но сформулирован в терминах графа потока управления [10].

системы[2]. Метод не зависит от языка программирования, использованного для разработки тестируемого программного обеспечения, не требует доступа к исходному коду и не предполагает инструментирования кода программного обеспечения.

3.1. Системные вызовы как характеристика поведения программы

Системными вызовами (*system call*) называются функции, входящие в состав программного интерфейса операционной системы. Изначально последовательности системных вызовов были предложены в качестве характеристики поведения программы при решении задачи обнаружения вторжений [2,3,4,5] в области компьютерной безопасности. Последовательности системных вызовов также успешно использовались при поиске потенциальных ошибок в программном обеспечении [17]. При этом было отмечено, что последовательности системных вызовов, произведенных программой во время своего выполнения, являются достаточно информативной характеристикой выполнения программы и позволяют различать, когда выполнение программы идет по разным путям [2]. В данной работе это свойство последовательностей системных вызовов используется для анализа поведения программы на наборе тестов.

Мониторинг системных вызовов, вызываемых программой во время своего выполнения, осуществляется командой *ktrace*, входящей в пакет операционной системы. Результатом работы этой программы является трасса системных вызовов, выполненных программой во время своей работы. На Рис.1 приведен пример трассы для фрагмента программы.

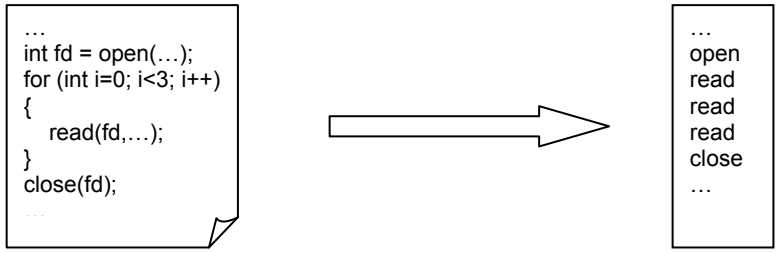

Рис.1 Представление поведения программы с помощью системных вызовов.

[2] Хотя в данной работе рассматривается случай операционной системы семейства UNIX, в состав которой входит команда мониторинга системных вызовов, предлагаемый метод может быть распространен на любую операционную систему, имеющую аналогичные средства мониторинга. Например, метод также может применяться в операционных системах семейства Linux, где для сбора трасс системных вызовов существует команда *strace*.

3.2. Модель поведения программы на тесте

В предлагаемом методе последовательности системных вызовов используются для построения модели поведения программы на тесте. Для начала введем несколько определений.

Трассой системных вызовов программы P для теста t будем называть последовательность системных вызовов, совершенных программой P при выполнении на входных данных тестового случая t. При этом будем подразумевать, что системные вызовы встречаются в трассе в порядке их вызова программой P[3].

Последовательностью системных вызовов длины K называется любая непрерывная подпоследовательность длины K, встречающаяся в трассе системных вызовов.

Множеством последовательностей системных вызовов длины K, соответствующим поведению программы P на тесте t, называется множество всех возможных последовательностей длины K, встречающихся в трассе системных вызовов программы P для теста t.

Для получения множества последовательностей системных вызовов фиксированной длины используется техника «скользящего окна» [3] с размером K (размер окна соответствует длине выделяемых последовательностей). Согласно этому подходу, выделение последовательностей происходит следующим образом: в качестве первой последовательности из трассы выбираются K идущих подряд системных вызовов, начиная с первого вызова в трассе; в качестве второй последовательности выбираются K идущих подряд системных вызовов, начиная со второго вызова; и так далее, пока не будет пройдена вся трасса.

В качестве *модели поведения* программы P на тесте t принимается определенное выше множество последовательностей.

Рассмотрим пример построения модели поведения программы. Допустим, у нас есть следующая последовательность, состоящая из системных вызовов операционной системы UNIX:

```
open write write open write close write close.
```

Тогда результатом выделения последовательностей с помощью скользящего окна размера 4 будет следующее множество последовательностей:

```
open write write open
write write open write
write open write close
open write close write
write close write close
```

[3] Команда *ktrace* предоставляет такую возможность.

Полученное таким образом множество последовательностей будет являться моделью поведения программы.

3.3. Метод сокращения набора тестов

Идея предлагаемого метода сокращения набора тестов достаточно проста. Можно предположить, что при тестировании функциональности и структуры программы встречаются тесты, которые инициируют одни и те же последовательности системных вызовов. Метод основан на этом предположении и состоит в «отсеивании» тех тестов, которые не генерируют новые последовательности системных вызовов, то есть не инициируют нового поведения программы.

В предлагаемом методе сокращения набора тестов используется модель поведения программы, построенная в предыдущем разделе. Общий алгоритм работы метода выглядит следующим образом (см. Рис.2):

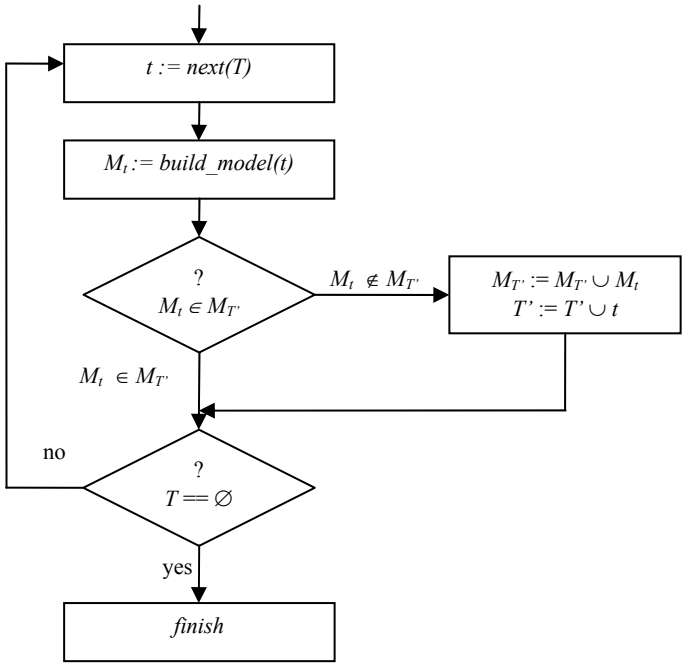

Рис.2 Алгоритм работы метода.

Здесь T – первоначальный набор тестов; T' – сокращенный набор тестов, $T' \subseteq T$; t – очередной тест из первоначального набора, $t \in T$; M_t – модель поведения программы на тесте t; $M_{T'}$ – множество моделей поведения программы на тестах из T'.

1. На первом этапе выбирается очередной тест из сокращаемого набора тестов и на нем выполняется тестируемая программа.
2. Во время выполнения программы строится модель M_t ее поведения на выбранном тесте t.
3. На третьем этапе проверяется, входит ли модель M_t в множество моделей $M_{T'}$ или нет.
4. Если модель M_t не входит в множество моделей $M_{T'}$, то она добавляется в множество $M_{T'}$, и тест t добавляется в сокращенный набор тестов T'.
5. Если набор тестов T не исчерпан, то осуществляется переход к шагу 1, в противном случае сокращенный набор тестов T' считается построенным, и работа метода завершается.

Для того чтобы реализовать метод, необходимо предложить операцию сравнения моделей поведения программы. Такая операция должна показывать, когда модели поведения программы совпадают, а когда нет. Будем считать, что модели поведения программы равны, когда их множества последовательностей совпадают, и различны в противном случае. Тогда операция сравнения моделей будет сводиться к операции сравнения множеств последовательностей системных вызовов.

Надо заметить, что использование в предложенном методе множества последовательностей системных вызовов вместо трасс системных вызовов позволяет решить проблему циклов в тестируемой программе. Данная проблема изначально возникла при оценке адекватности наборов тестов, основанной на анализе покрытия путей выполнения программы, и состоит в том, что если код программы содержит циклы, то количество различных путей выполнения программы может быть очень большим, что приводит к падению эффективности метода оценки и значительному увеличению затрат на его работу [15]. Аналогичная проблема возникает и в рассматриваемом методе, если в качестве модели поведения программы рассмотривать трассы системных вызовов. В этом случае, если системные вызовы встречаются внутри циклов программы, различное количество возможных повторений цикла может привести к очень большому количеству разных трасс программы. Такая ситуация может исказить результаты сокращения тестового набора, так как поведения программы, различающиеся только количеством выполненных циклов, будут приняты как различные, хотя фактически они не тестируют новую функциональность или структуру программы. Использование множества последовательностей системных вызовов решает эту проблему, так как, даже при произвольном количестве повторений цикла, множество уникальных поведений программы быстро стабилизируется, и модель перестает изменяться.

4. Экспериментальное исследование метода сокращения набора тестов

Целью проводимого эксперимента являлось определение и оценка параметров метода сокращения набора тестов, предложенного в данной работе, на модельных данных.

При анализе методов сокращения набора тестов важны следующие параметры:

1. Степень сокращения набора. Это один из наиболее существенных параметров работы метода сокращения, так как этот параметр показывает, насколько сокращаются первоначальный набор тестов и связанные с ним затраты. Чем больше степень сокращения набора тестов, тем сильнее уменьшаются затраты на его выполнение, проверку и сопровождение, тем самым, сокращая затраты на проведение регрессионного тестирования в целом [10].
2. Уровень обнаружения ошибок. Основной проблемой при сокращении набора тестов является возможное уменьшение его уровня обнаружения ошибок вследствие удаления тестов, обнаруживающих ошибки [10]. Слишком большие потери в уровне обнаружения ошибок могут привести к нецелесообразности сокращения набора тестов. Поэтому важно, чтобы в результате работы метода его уровень обнаружения ошибок по возможности не уменьшался, а если и уменьшался, то незначительно.

Целью эксперимента являлось решение следующих задач:

1. определение величины сокращения исходного набора тестов;
2. определение уровня обнаружения ошибок сокращенным набором тестов;
3. сравнение уровня обнаружения ошибок сокращенным набором тестов с уровнем первоначального набора и уровнем, обеспечиваемым методом случайного сокращения набора тестов;

4.1. Условия проведения эксперимента: тестовые программы и тестовые наборы

Эксперимент проводился в операционной системе FreeBSD 5.3; для сбора трасс использовалась команда *ktrace,* входящая в состав операционной системы. Длина K используемых последовательностей составляла 15 системных вызовов.

Для проведения эксперимента в качестве объекта тестирования использовались программа *nametbl*, входящая в пакет программ, впервые предложенный в [6] и впоследствии использовавшийся в [7,8,9] для исследования методов оценки адекватности тестовых наборов. Программа *nametbl* считывает команды из файла и выполняет их с целью тестирования нескольких функций. Тестируемые функции реализуют работу с таблицей символов для некоторого языка программирования. Таблица символов хранит для каждого символа его имя и тип. Команды, считываемые из файла,

позволяют добавить в таблицу новый символ, указать тип ранее введенного символа, осуществить поиск символа и распечатать таблицу символов. Общий объем исходного текста программы составляет 356 строк. Основной причиной выбора этой программы для проведения эксперимента послужило то, что ее исходные тексты и руководство по созданию наборов тестов находятся в открытом доступе[4], и, таким образом, описываемый эксперимент может быть повторен независимо.

Для построения тестов использовалось разбиение входных данных программы *nametbl* на домены, предложенное в работе [9]. В этой работе было предложено 4 «измерения» входных данных, в трех из которых было выделено 7 доменов, а в четвертом – 5 доменов. В данной работе было использовано это разбиение, в каждом домене было выбрано по одному значению, и в итоге был сгенерирован пул из 1715 разных тестов.

Для определения параметров метода были использованы следующие индикаторы:

1. размер сокращенного набора тестов;
2. коэффициент сокращения набора тестов в процентном выражении: $100 \times (1 - Size_{reduced}/Size_{initial})$, где
   $Size_{initial}$ означает размер исходного набора тестов,
   $Size_{reduced}$ – размер сокращенного набора тестов;
3. количество обнаруженных ошибок;
4. коэффициент обнаружения ошибок в процентном выражении: $100 \times (FaultsDetected_{reduced}/FaultsDetected_{initial})$, где
   $FaultsDetected_{initial}$ означает количество ошибок, обнаруживаемых исходным набором тестов,
   $FaultsDetected_{reduced}$ – количество ошибок, обнаруживаемых сокращенным набором тестов.

Уровень обнаружения ошибок измерялся исходя из следующего принципа эквивалентности тестовых наборов: два набора тестов считаются эквивалентными при обнаружении некоторой ошибки, если каждый набор содержит по крайней мере один тест, обнаруживающий эту ошибку (в англоязычной литературе такой принцип носит название *per-test-suite basis* [10]). Такой подход применялся в работах [1, 10].

4.2. Проведение эксперимента и результаты

Так как стратегия генерации тестовых наборов оказывает косвенное влияние на исследуемый метод (вследствие того, что индивидуальные тесты выбираются из набора в порядке их следования, и, соответственно, в сокращенный набор тестов будет помещен первый тест, расширяющий

[4] Исходный текст и документация к программе доступны на сайте в интернете по адресу: http://www.chris-lott.org/work/exp/.

множество моделей), то для достижения чистоты эксперимента тесты исходного набора были перенумерованы случайным образом.

После этого тестируемая программа была выполнена на каждом тесте из пула, и для каждого теста была построена модель выполнения программы. Затем из пула сгенерированных тестов методом случайной выборки было сформировано 35 исходных наборов тестов разного размера, начиная с 50 и заканчивая 1700 с шагом 50. В качестве последнего, 35-го набора был принят набор, содержащий все 1715 тестов из пула. Далее, из исходных наборов с помощью исследуемого метода были построены сокращенные наборы, для каждого из которых были вычислены параметры сокращения: размер сокращенных наборов и коэффициент сокращения. После этого были вычислены уровни обнаружения ошибок сокращенными наборами, и затем результаты были сравнены с уровнями исходных наборов и уровнями, получаемыми при использовании метода случайного сокращения. Для этого из исходных наборов методом случайной выборки были построены наборы такого же размера, как и сокращенные, и были вычислены их уровни обнаружения ошибок.

4.3. Результаты эксперимента

Результаты работы метода проиллюстрированы на Рис. 3 – 6:

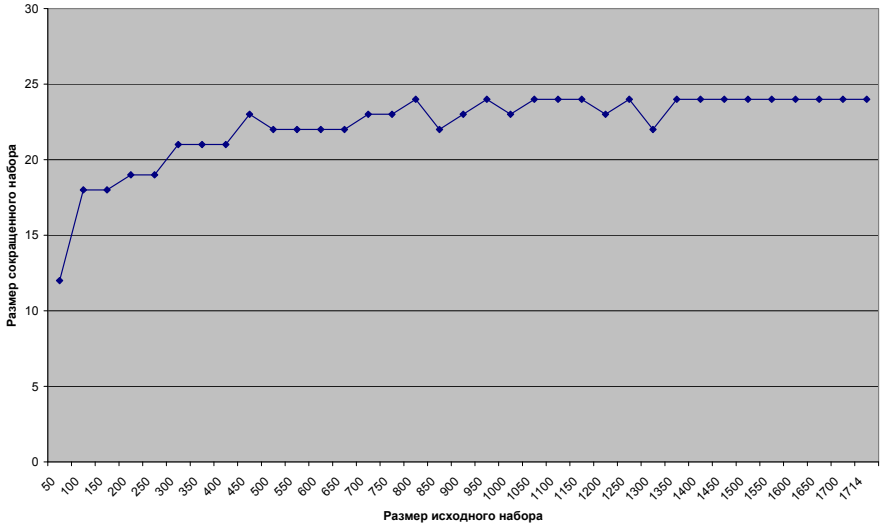

Рис. 3 Размер сокращенных наборов тестов.

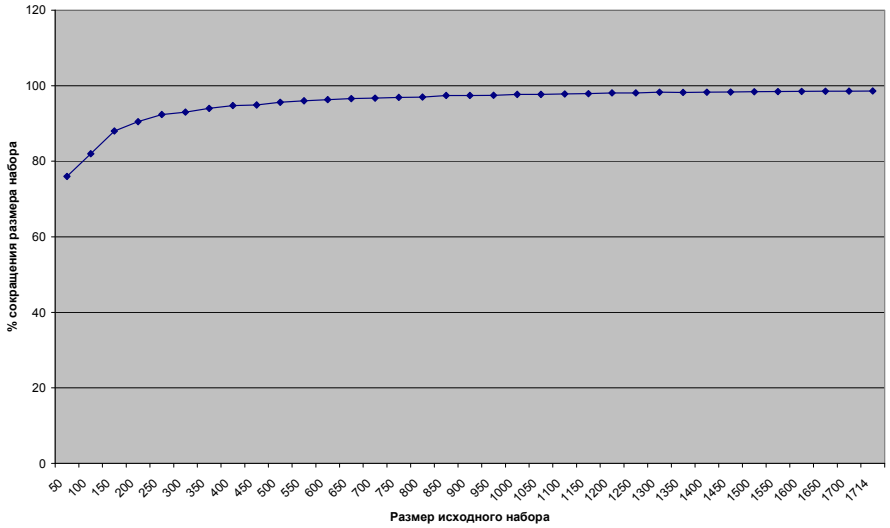

Рис. 4 Процент сокращения набора тестов.

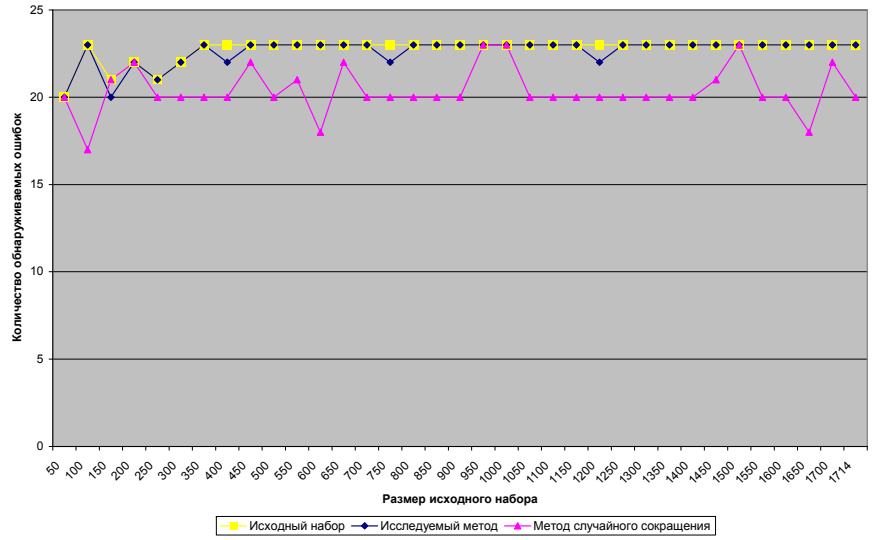

Рис. 5 Количество ошибок, обнаруживаемых наборами тестов.

Размер сокращенного набора находился в пределах от 12 до 24 тестов в зависимости от размера исходного набора. Величина сокращения находилась

в пределах от 76% до 98.6% в процентном соотношении. По результатам видно, что размер сокращенного набора рос вместе с ростом размера исходного набора и в итоге стабилизировался. Это является закономерной картиной, если учесть, что множество различных путей выполнения программы конечно, и при увеличении числа тестов увеличивается вероятность выполнения всех путей программы.

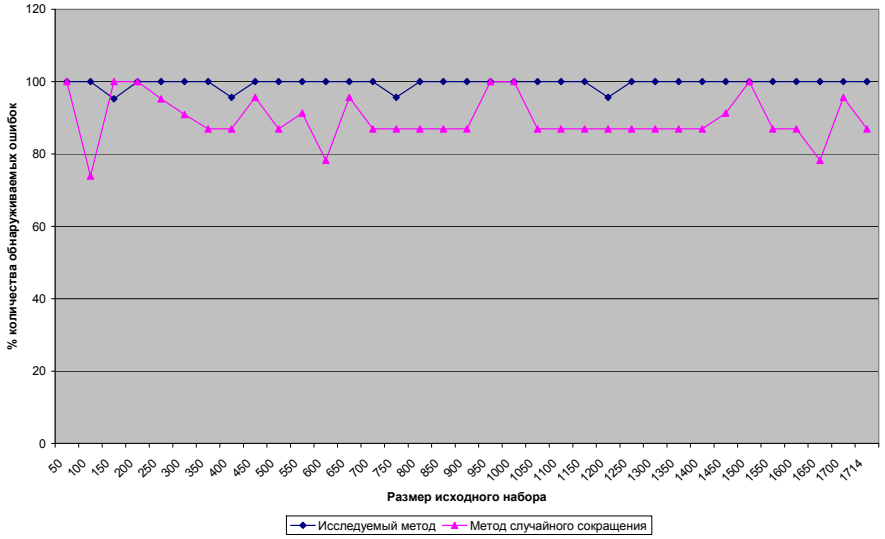

Рис. 6 Процент обнаружения ошибок наборами тестов.

На модельных данных уровень обнаружения ошибок для сокращенных наборов составил от 95.24% до 100% от уровня исходных наборов, т.е. величина падения уровня обнаружения ошибок для исследуемого метода составила менее 4.8%. Средняя величина падения уровня обнаружения ошибок для исследуемого метода составила 1.5%. При этом исследуемый метод проиграл методу случайного сокращения лишь в одном случае; в большинстве остальных случаев исследуемый метод показал выигрыш на 4.3% - 26%. В среднем сокращенные наборы, построенные с помощью исследуемого метода, при проведении эксперимента обнаруживали на 9.8% ошибок больше, чем соответствующие наборы, полученные с помощью метода случайного сокращения.

5. Заключение и направление будущих исследований

В данной работе был представлен метод решения задачи сокращения набора тестов, возникающей в контексте регрессионного тестирования. Данный метод основан на использовании моделей поведения программы, построенных

в терминах системных вызовов, которые производятся программой во время своего выполнения. Работа метода была экспериментально исследована на модельных данных. Были исследованы такие параметры работы метода, как величина сокращения набора тестов и уровень обнаружения ошибок. Результаты работы метода были сравнены с результатами метода случайного сокращения набора тестов, и было показано, что исследуемый метод практически не уступает методу случайного сокращения набора тестов, а в большинстве случаев превосходит его.

В будущих исследованиях планируется развивать предложенный метод и, в частности, сосредоточиться на следующих направлениях. Во-первых, необходимо более четко охарактеризовать класс программ, к которым может быть применен предложенный метод. Интуитивно понятно, что результаты работы метода зависят от внутреннего устройства программы – частоты и характера обращений тестируемой программы к системным вызовам операционной системы. Например, на преимущественно вычислительных программах, где обращения к системным вызовам происходят редко, или в программном обеспечении баз данных, где разнообразность поведения программы характеризуется скорее характером передаваемых данных, нежели разными путями в графе выполнения программы, исследуемый метод может и не показать хороших результатов. Во-вторых, имеет смысл учитывать не только имена системных вызовов, совершенных программой во время своего выполнения, но и передаваемые в них параметры и возвращаемые значения. Такая информация косвенно учитывает потоки данных в программе и будет полезна при построении модели поведения программы. Наконец, планируется сравнить результаты работы предлагаемого метода не только с методом случайного сокращения наборов тестов, но и с остальными методами, используемыми в данной области.

Литература

[1] McMaster, S. and Memon, A. M. 2005. Call Stack Coverage for Test Suite Reduction. In Proceedings of the 21st IEEE international Conference on Software Maintenance (ICSM'05) - Volume 00 (September 25 - 30, 2005). ICSM. IEEE Computer Society, Washington, DC, 539-548.

[2] S.Forrest, S.A.Hofmeyr, A.Somayaji, and T.A.Longstaff. A Sense of Self for Unix Processes. In Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[3] S.A.Hofmeyr, S.Forrest, and A.Somayaji. Intrusion Detection using Sequences of System Calls. Journal of Computer Security, 6:151–180, 1998.

[4] C. Warrender, S. Forrest, and B. Pearlmutter, Detecting Intrusions Using System Calls: Alternative Data Models. IEEE Symposium on Security and Privacy, Oakland, CA, 1999, IEEE Computer Society, pp. 133-145.

[5] Eleazar Eskin, Andrew Arnold, Mechael Prerau, Leonid Portnoy, Salvatore J. Stolfo. A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. Applications of Data Mining in Computer Security, Kluwer, 2002.

[6] Victor R. Basili and Richard W. Selby. Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering, SE-13(12):1278-1296, December 1987.

[7] E. Kamsties and C.M. Lott, An Emperical Evaluation of Three Defect-Detection Techniques, In: W. Schfer and P. Botella (eds.). Software Engineering - ESEC '95, Springer, 1995, pp. 362-383.

[8] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques:a replicated empirical study. In H. S. Mehdi Jazayeri, editor, Proceedings of The Sixth European Software Engineering Conference, volume 1301, pages 262-277. Lecture Notes in Computer Science, September 1997.

[9] M. Grindal, B. Lindstrom, A.J. Offutt, and S.F. Andler. An Evaluation of Combination Strategies for Test Case Selection, Technical Report. Technical Report HS-IDA-TR-03-001, Department of Computer Science, University of Skovde, 2003.

[10] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. Journal of Software Testing, Verification, and Reliability, V. 12, no. 4, December, 2002.

[11] X. Ma, Z. He, B. Sheng, C. Ye. A Genetic Algorithm for Test-Suite Reduction. 2005 IEEE International Conference on Systems, Man and Cybernetics. 10-12 Oct. 2005, Vol. 1, pp: 133- 139.

[12] J. A. Jones, M. J. Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, Vol. 29, No. 3, March, 2003, pp. 195-209.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology, 2(3):270-285, July 1993.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering, pages 191-200, May 1994.

[15] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol.29, No.4, pp.366-427, December 1997.

[16] Code Coverage Analysis. http://www.bullseye.com/coverage.html, 2002.

[17] V. Dallmeier. Detecting Failure-Related Anomalies in Method Call Sequences. Diploma Thesis, Universität des Saarlandes, March, 2005.