ПРИМЕНЕНИЕ ЭВОЛЮЦИОННЫХ АЛГОРИТМОВ ДЛЯ ПОКРЫТИЯ КОДА ТЕСТАМИ

М. В. Буздалов

ассистент кафедры компьютерных технологий;
mbuzdalov@gmail.com

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Аннотация: В данной работе описываются основные положения применения эволюционных алгоритмов для покрытия кода тестами. Описывается новый вид функции приспособленности теста относительно еще не покрытого фрагмента кода и метод использования такой функции в эволюционных алгоритмах с большим числом особей в поколении.

Введение

Тестирование программного обеспечения (ПО) занимает до 50% времени и более 50% стоимости производства ПО [1]. По этой причине, автоматизация тестирования ПО, за счет уменьшения затрачиваемого времени и снижения влияния человеческого фактора, приводит к сокращению периода выхода программного продукта на рынок, снижению его себестоимости и уменьшению числа программных ошибок в нем. Такие методологии производства ПО, как экстремальное программирование [2], уделяют значительное внимание написанию так называемых модульных тестов, а также автоматизации их регулярного запуска, однако, написание модульных тестов выполняется программистами, а следовательно, этот процесс нуждается в усовершенствовании. В частности, актуальной задачей является автоматическая генерация тестов.

Одним из показателей качества набора тестов является то, насколько полно, в смысле возможных путей выполнения и возможных случаев, встречающихся в коде, тестируемый код покрыт тестами. Существует множество критериев покрытия кода тестами, как общих, так и специализированных. Примером общего критерия является число (доля) покрытых строк кода (то есть, число таких строк, которые хотя бы один раз исполнялись при тестировании кода на рассматриваемом наборе тестов). Примером специализированного критерия для тестирования баз данных SQL является доля покрытых тестами операторов SELECT.

Цель работы

В работе рассматривается автоматическая генерация набора тестов, покрывающего код согласно заданному критерию, с применением эволюцион-

ных алгоритмов [3, 4]. Рассматриваются как широко распространенные критерии покрытия кода, так и новые критерии, ориентированные на покрытие значений, в особенности — граничных значений.

Критерии покрытия кода тестами

Существует множество критериев покрытия кода тестами. Перечислим наиболее известные из них:

- покрытие путей;
- покрытие условий;
- покрытие ветвлений;
- покрытие строк кода;
- покрытие методов (функций, процедур);
- покрытие классов.

Покрытие путей — наиболее точный, но практически мало используемый на практике критерий, поскольку в большинстве случаев выполнение тестируемого кода может идти по бесконечному или зависящему экспоненциально от размера кода числу различных путей. Покрытие методов и покрытие классов — достаточно простые (для реализации стопроцентного покрытия) критерии, но, как правило, они не отражают реальную протестированность кода. Покрытие ветвлений — это облегченный вариант покрытия условий. Разницу между этими двумя критериями можно проиллюстрировать с помощью листинга 1.

Листинг 1

Иллюстрация покрытия условий и покрытия ветвлений

```
function a(x, y, z) {
    if (x && y && z) {
        a();
    } else {
        b();
    }
}
```

Покрытие ветвлений (каждая ветвь кода выполнена не менее одного раза) можно достичь двумя тестами:

- `x = y = z = false;`
- `x = y = z = true.`

В то же время покрытие условий потребует как минимум четырех тестов, что позволяет тщательнее проверить корректность условия ветвления:

- `x = y = z = true;`
- `x = y = true, z = false;`
- `x = true, y = z = false;`
- `x = y = z = false.`

Кроме данных критериев покрытия, можно ввести специальные критерии, которые призваны проверять различные крайние и граничные случаи, не обрабатываемые в явном виде с помощью операторов ветвления, например:

- минимальные и максимальные значения;
- значения `null` и не-`null` для ссылочных типов;
- пустые, одноэлементные и многоэлементные структуры данных;
- целочисленное деление на ноль;
- экстремальные значения аргументов функций (например, $\mathrm{arctg}(\pi/2)$);
- значения каждого бита в битовой маске;
- ожидаемые и неожидаемые значения аргумента оператора побитового сдвига.

Такое *покрытие значений* может быть особенно полезным при тестировании кода, работающего с математическими или битовыми операциями, а также со сложными структурами данных. Задача о покрытии тестами кода с данными свойствами часто встречается при подготовке тестов для олимпиадных задач по программированию.

Расстояние до ветви и до значения

Расстояние до ветви — это мера того, насколько требуется изменить входные данные для того, чтобы данная ветвь кода была выполнена. Для каждого типа условий требуется разработать новую функцию, задающую расстояние до ветви. Простейший пример приведен в листинге 2.

Листинг 2

Оператор ветвления и соответствующее расстояние до ветви

```
if (a * 72 > 1336) {
    //Требуемая ветвь
}
```

```
d(a)= 1336 - a * 72
```

Для более сложных условий ветвления функции расстояния до ветви строятся соответствующим образом, при этом может понадобиться профи-

лирование или модификация библиотечного кода (листинг 3). Аналогичным образом можно строить функции *расстояния до значения*.

Листинг 3

Более сложные примеры условий ветвления

```
if (str.startsWith("example")) { /* ... */ }
if (str.matches("[0..9]+")) { /* ... */ }
```

На основе функций ветвления далее будут строиться функции приспособленности, описывающие то, насколько тест близок к покрытию того или иного фрагмента кода или значения переменной.

Описание предлагаемого подхода

Предлагаемый подход к генерации покрывающих тестов принадлежит методологии «тестирования белого ящика», то есть, требует доступа к коду тестируемой программы (исходных кодов или исполняемого кода). Он состоит из двух этапов. На первом этапе, в соответствии с конечной целью генерации набора тестов, выбирается критерий покрытия. Может быть выбран как один из традиционных критериев, такой как доля покрытых инструкций кода или доля покрытых ветвей кода, так и какой-либо другой критерий, который лучше подходит для поставленной задачи.

Второй этап состоит в генерации тестов. Для этого вначале строится несколько случайных тестов, тесты добавляются в итоговый набор, и анализируется полученное ими покрытие. Затем для некоторого непокрытого фрагмента программы, согласно выбранному критерию, строится оптимизируемая функция (или, в терминах эволюционных алгоритмов, функция приспособленности). Функция вычисляется при выполнении тестируемой программы путем модификаций ее исходного кода. В данной работе она имеет сложный вид, зависящий от последовательности выполненных операций и значений, принимаемых переменными в процессе выполнения. Чем «ближе» траектория выполнения программы на тесте подходит к выбранному фрагменту, тем «лучше» значение функции приспособленности.

Для оптимизации указанной функции производится запуск эволюционного алгоритма. Особью алгоритма является тест. Когда фрагмент оказывается покрытым некоторым тестом, этот тест добавляется в итоговый набор, затем вновь анализируется покрытие программы текущим итоговым набором, и описанная процедура повторяется, пока покрытие не будет признано удовлетворяющим поставленной задаче.

Функции приспособленности

В литературе [5, 6] встречаются, главным образом, два варианта задания функции приспособленности теста по отношению к еще не покрытому

фрагменту кода. Первый из них состоит в задании предопределенного пути в графе управления тестируемого кода. Функция приспособленности состоит из двух компонент — длины общего префикса заданного и фактического пути и расстояния до непосещенной ветви в узле, где эти пути расходятся.

Второй вариант отличается от первого тем, что расстояние до непосещенной ветви измеряется не в первом узле, где пути разошлись, а в ближайшем таком узле к цели покрытия. Данная модификация мотивируется тем, что запланированный и фактический пути выполнения после расхождения могут вновь объединяться.

Автором настоящей работы предлагается новый метод, состоящий в том, что путь до целевого узла не задается. Значения функции приспособленности при этом подходе состоят из идентификатора ближайшего к целевому узла, принадлежащего фактическому пути, и значения расстояния до непосещенной ветви в этом узле. При этом сравнимыми оказываются не все такие значения. Таким образом, можно рассматривать аналог Парето-фронта для тестов, приближающегося к цели покрытия по мере улучшения составляющих его тестов. Данный метод призван использовать параллелизм исследования пространства поиска, присущий эволюционным алгоритмам, работающим с поколениями из большого числа особей, таким как генетические алгоритмы. Можно отметить общие черты описанного метода с алгоритмами многокритериальной оптимизации.

Литература

1. *Myers G.* The Art of Software Testing, Second Edition. John Wiley & Sons, 2004.
2. *Бек К.* Экстремальное программирование. Питер, 2002.
3. *Mitchell M.* An Introduction to Genetic Algorithms. MIT Press, 1996.
4. *Luke S.* Essentials of Metaheuristics. Lulu, 2009.
5. *Goldberg A., Wang T. C.,* and *Zimmerman D.* Applications of feasible path analysis to program testing // In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 1994.
6. *Baars A., Harman M* et al. Symbolic Search-Based Testing // In Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering, 2011.