ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук
Департамент информатики

Эгипти Павел Андреевич

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ МОДУЛЬНЫХ ТЕСТОВ ДЛЯ ЯЗЫКА GO

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Руководитель
кандидат физико-математических наук,
доцент, департамент информатики,
М.С. Мухин

Рецензент
ООО «Техкомпания Хуавэй»,
руководитель отдела,
Д.С. Фокин

Санкт-Петербург
2023

Оглавление

Аннотация

Для того, чтобы проверить корректность определенных модулей кода программы, существует модульное тестирование. Программисты пишут модульные тесты для функций и методов для их дальнейшего запуска и проверки на наличие ошибок в коде. Модульные тесты необходимы для отслеживания появления регрессии при изменении исходного кода. Они являются лучшим средством для фиксации поведения кода. Но существует ряд проблем, связанных с написанием модульных тестов.

Решение этих проблем и существующие решения из смежных с модульным тестированием других видов тестирования натолкнули на идею генерации модульных тестов. Существует несколько решений для языка Go, автоматически генерирующих модульные тесты, но некоторые из них имеют весьма сильные ограничения на применение, а другие плохо справляются со своей задачей и демонстрируют низкие показатели по покрытию кода.

В данной работе было предложено использовать фаззинг для генерации входных данных для функций. На его основе реализовано приложение для генерации модульных тестов. Также были проведены сравнения с существующими аналогами.

Ключевые слова: модульные тесты, генерация тестов, фаззинг.

To ensure the correctness of specific code modules in a program, there is a technique called unit testing. Programmers write unit tests for functions and methods to run them and check for any errors in the code. Unit tests are essential for detecting regression issues when making changes to the source code, and they serve as an effective means of documenting code behavior. However, there are several challenges associated with writing unit tests.

Addressing these challenges and drawing inspiration from solutions in other types of testing led to the idea of generating unit tests. Several solutions exist for the Go programming language that automatically generate unit tests, but some of them have significant limitations, while others struggle with their intended purpose and demonstrate low code coverage.

In this work, the proposal was made to leverage fuzzing techniques to generate input data for functions. Based on this approach, an application for generating unit tests was implemented. Furthermore, comparisons were conducted against existing alternatives.

Keywords: unit tests, test generation, fuzzing.

Введение

Модульное тестирование является неотъемлемой частью процесса разработки программного обеспечения. Стоит уделять должное внимание этому процессу на ранних стадиях разработки, так как это позволяет упростить ее в дальнейшем, а также позволяет обнаружить существующие ошибки, пока они не сказались на конечном пользователе. Поэтому важно писать модульные тесты.

Модульный тест — это фрагмент кода, который вызывает другой фрагмент кода и впоследствии проверяет правильность некоторых предположений [10]. Если предположения оказываются неверными, модульный тест не пройден. Модульные тесты обладают большим набор преимуществ:

1. Помогают заниматься рефакторингом кода. Программист перед тем, как приступить к рефакторингу, запускает тесты и проверяет, что они проходят, после этого он упрощает код, делает его более понятным и расширяемым. Дальше запускает тесты и проверяет, что они все так же проходят.

2. Являются лучшим средством фиксации поведения кода.

3. Являются хорошим видом документации кода. Модульные тесты постоянно запускаются на CI (Continuous Integration), что не дает им устаревать.

Но написание хороших модульных тестов связано с существованием ряда проблем. Во-первых, программисты не любят писать тесты. Во-вторых, программисты при написании тестов зачастую пристрастны и предвзято относятся к своему коду, что в конечном итоге негативно сказывается на количестве рассмотренных крайних случаев. В-третьих, написание хороших модульных тестов может быть времязатратной и рутинной задачей. Из-за существующих проблем и желания упростить жизнь программисту возникла идея автоматической генерации модульных тестов.

На данный момент существует большое количество генераторов модульных тестов для разных языков программирования с использованием разных техник. Например:

- UnitTestBot - семейство проектов, разрабатываемых для автоматической генерации модульных тестов и анализа кода для таких языков, как Java/Kotlin, C/C++, Python и JavaScript.
- Randoop - генератор модульных тестов для Java.
- Evosuite - генератор наборов тестов JUnit для классов Java.

Для языка Go тоже есть генераторы модульных тестов. Но те технологии, которые они используют не позволяют им добиться хороших результатов в покрытии, либо накладывают сильные ограничения на применимость таких генераторов. Среди ограничений можно выделить ограничения на типы данных, существующих в языке Go, использование внешних и стандартных библиотек и др.

Главной задачей при разработке генератора модульных тестов является выбор техники для генерации тестовых входных данных для функций. Возможно, самым простыми методом является случайная генерация входных данных. Более развитая форма, называемая фаззингом, состоит в том, чтобы начать с правильно сформированных входных данных и неоднократно изменять их, более или менее случайным образом, для получения новых входных данных. Это оказалось эффективным способом поиска сбоев и уязвимостей безопасности в программном обеспечении, причем некоторые из самых известных уязвимостей безопасности были обнаружены иммено таким образом [4]. Фаззинг стал стандартным элементом большинства коммерческих стратегий тестирования программного обеспечения [1, 7, 9]. Еще одной техникой является символьное исполнение. Но ее применение связано с решением многих проблем такого подхода, что в свою очередь накладывает множество ограничений на использование генератора пользователем.

Цели и задачи

Данная работа ставит целью создание генератора модульных тестов для языка Go с использованием фаззинга. Для достижения поставленной цели необходимо решить следующие задачи:

- Выбрать фаззер для генерации тестовых входных данных для функций.
- Реализовать инструмента для автоматической генерации модульных тестов и интегрировать его в UnitTestBot для языка Java.
- Сравниться с аналогами в покрытии кода тестами.

Достигнутые результаты

В рамках данной работы были проанализированы существующие фаззеры на Go. В качестве фаззинга была выбрана фаззинг платформа из проекта UnitTestBot для языка Java. На ее основе был создан coverage-guided fuzzer. Был реализован генератор модульных тестов для языка Go:

- Поддерживается генерация тестов для функций и методов.
- Поддерживаются все типы, кроме функций.

Генератор был интегрирован в UnitTestBot для языка Java:

- Создано приложение командной строки для демонстрации работы генератора на веб-сайте utbot.org.
- Плагины для IntelliJ IDEA Ultimate и GoLand интегрированы в плагин UnitTestBot для языка Java.

Также были проведены сравнения с аналогами в покрытии кода тестами на простых и сложных функциях.

Структура работы

В главе 1 описаны существующие техники для автоматической генерации тестов.

В главе 2 описаны существующие фаззеры для языка Go.

В главе 3 описаны существующие решения, позволяющие автоматически генерировать модульные тесты для языка Go.

В главе 4 описана архитектура приложения и все этапы его работы.

В главе 5 приведены сравнения полученного решения с существующими аналогами.

В заключении представлено краткое описание достигнутых результатов по реализации генератора модульных тестов для языка Go.

1. Обзор техник для автоматической генерации тестов

В данной главе будут рассмотрены следующие техники, используемые для генерации тестов:

- Случайное тестирование
- Фаззинг тестирование
- Тестирование на основе поиска
- Символьное исполнение

1.1. Случайное тестирование

Случайное тестирование является самым простым способом тестирования программного обеспечения, так как входные данные для программ генерируются случайно и независимо, что позволяет экономить на процессе генерации новых данных и производить большое число тестовых запусков за определенное время.

Главным плюсом такой техники является простота ее реализации, а главным минусом нахождение только основных ошибок.

1.2. Фаззинг тестирование

Фаззинг - это тестирование программ на неправильных, неожиданных или случайных входных данных.

Фаззером называется программа, проводящая тестирование с использованием фаззинга.

Существует несколько способов классификации фаззера:

- На основе генерации или на основе мутаций (модификаций) — в зависимости от того, новые данные генерируются или получаются путем изменения существующих входных данных. Фаззеры на основе генерации часто создают новые входные данные случайным

образом, фаззеры на основе мутаций берут существующие входные данные и мутируют (модифицируют) их. Также существуют фаззеры, которые могут совмещать эти два подхода.

- Dumb или smart - в зависимости от того, знает ли фаззер структуру ввода. Dumb фаззеры не требуют входной модели. Например, AFL [13] — dumb фаззер на основе мутаций, который модифицирует начальный вход, изменяя случайные биты, заменяя случайные биты «интересными» значениями, а также перемещая или удаляя блоки данных. Smart фаззеры, наоборот, используют входную модель для генерации входных данных.

- Белый, серый или черный ящик - в зависимости от того, знает ли фаззер о структуре программы [2, 6]. Фаззер белого ящика анализирует внутреннюю структуру программы. Но для этого необходимо, как сказано выше, иметь информацию о внутренней структуре программы, которая не всегда доступна. Кроме того, методы, которые используются в фаззинге белого ящика, очень трудоемки, где часто необходимы SMT-решатели [2]. Черный ящик — полная противоположность фаззеру белого ящика, он не знает о внутренней структуре программы и не получает никакой информации от программы. Большинство существующих фаззеров относятся к этой категории [2]. Они просты в создании и распараллеливании, в итоге такие фаззеры могут выполнять огромное количество запусков программы на случайных или измененных входных данных. Однако на самом деле фаззеры черного ящика не могут найти большинство существующих ошибок, потому что результат их работы сильно зависит от рандома. Фаззер серого ящика является чем-то средним. между двумя другими. Он может получать некоторую информацию о выполнении программы или о ее внутренней структуре. Например, AFL и libFuzzer [8] собирают некоторую информацию о путях выполнения программы.

Существует два варианта повышения качества работы фаззера [5]. Первым вариантом является увеличение числа запусков программы за единицу времени путем уменьшения количества времени, требуемого на генерацию новых входных данных, а вторым - улучшение качества сгенерированных данных путем получения и анализа обратной связь от предшествующих выполнений, чтобы направить генерацию входных данных на те входные данные, которые с большей вероятностью обнаружат ошибки. Для получения обратной связи фаззеры применяют инструментацию программ, что сказывается на ресурсах процессора.

В идеале одновременно увеличивать количество запусков программы за единицу времени и улучшать качество сгенерированных данных, но, к сожалению, это невозможно, так как они зависят друг от друга, поэтому зачастую приходиться выбирать, чему отдать предпочтение.

Фаззером, реагирующим на покрытие, называется coverage-guided fuzzer. Задачей таких фаззеров является покрытие новых участков кода. Принцип работы этих фаззеров весьма прост:

```
Собирается корпус входных данных (опциональный шаг)
for {
    Выбираются данные из корпуса
    Мутируются
    Запускается программа
    Собирается информация о покрытии
    if были покрыты новые участки кода {
        Данные добавляются в корпус
    }
}
```

Coverage-guided fuzzer-ы являются наиболее популярными из-за своей простоты и хороших результатов на практике. К таким фаззерам относятся уже упомянутые AFL и libFuzzer.

1.3. Тестирование на основе поиска

Еще одной техникой для генерации тестов является тестирование на основе поиска (search-based approach). Наиболее популярными алгоритмами, используемыми в этих целях, являются эволюционные алгоритмы [11]. Метрикой, определяющей меру приспособленности, выступает покрытие, максимизация которого ставится главной целью работы алгоритма.

1.4. Символьное исполнение

Символьное исполнение - это подход исполнения программ, который использует символьные значения вместо конкретных и исполняет множество путей исполнения программы вместо одного. Во время такого исполнения программы поддерживаются для каждого исследованного пути:

- Булева формула, которая описывает условия, которым удовлетворяют ветви, выбранные на этом пути
- Хранилище, которое отображает переменные в символьные выражения или значения

Средство проверки модели, обычно основанное на SMT-решателе [3], в конечном итоге используется для проверки того реализуем ли сам путь, т. е. может ли быть формула удовлетворена некоторым присвоением конкретных значений символическим аргументам программы. Такой подход позволяет пройтись по всем возможным путям исполнения программы.

2. Выбор фаззера

В этой главе был проведен анализ существующих фаззеров для языка Go, а также фаззинг платформы - универсального решения для генерации данных любого формата, используя фаззинг.

2.1. Обзор существующих фаззеров для языка Go

Ниже представлен обзор существующих фаззеров для языка Go: go-fuzz, Go Fuzzing и go-fuzz + gofuzz.

2.1.1. go-fuzz

go-fuzz[1] - coverage-guided fuzzing решение для тестирования пакетов. Это был самый популярный фаззер для Go до выхода нативного фаззера.

Чтобы им воспользоваться надо написать функцию со следующей сигнатурой:

```
func Fuzz(data []byte) int
```

Данные, приходящие на вход функции, являются сгенерированными go-fuzz. Функция должна возвращать 1, если фаззер должен повысить приоритет данного ввода во время последующего фаззинга; -1, если ввод нельзя добавлять в корпус, даже если он дает новое покрытие; и 0 в противном случае.

Среди ограничений можно выделить то, что функция должна лежать в любом пакете, кроме main, и, как видно из сигнатуры функции Fuzz, из типов поддерживается только []byte, который пользователю нужно самостоятельно преобразовывать в аргументы для тестируемой функции.

Также go-fuzz не дает доступ к информации о покрытии функции.

[1] go-fuzz: `https://github.com/dvyukov/go-fuzz`

2.1.2. Go Fuzzing

Go Fuzzing[2] - стандартный инструмент языка Go для фаззинга, который появился в версии языка 1.18.

Есть несколько правил, которые нужно соблюсти, чтобы воспользоваться этим фаззером:

1. Нужно написать специальную функцию, название которой должно начинаться с Fuzz. Среди параметров этой функции должен быть только параметр с типом *testing.F. При этом функция не должна ничего возвращать.

2. Функция должна лежать в файле *_test.go.

3. Функция для фаззинга должны быть передана в вызов метода (*testing.F).Fuzz. Она должна принимать *testing.T в качестве первого параметра, за которым следуют аргументы фаззинга, и не должна ничего возвращать.

4. (Опционально) Чтобы задать начальное значение корпуса, нужно передать значения тех же типов, что и аргументы функции для фаззинга, и в том же порядке в метод (*testing.F).Add.

5. Аргументы функции для фаззинга могут быть следующих типов: string, []byte, int, int8, int16, int32/rune, int64, uint, uint8/byte, uint16, uint32, uint64, float32, float64, bool

2.1.3. go-fuzz + gofuzz

Следующий фаззер можно получить путем использования пакета gofuzz[3] для преобразования массива байтов в объекты языка Go. Вообще, gofuzz был создан для создания Go объектов с случайными значениями. Но есть и второе его применение, как раз, для go-fuzz. Среди поддерживаемых типов - все, кроме каналов, интерфейсов и функций.

[2]Go Fuzzing: `https://go.dev/security/fuzz/`
[3]gofuzz: `https://github.com/google/gofuzz`

2.2. Фаззинг платформа

Фаззинг платформа - модуль проекта UnitTestBot для языка Java, разработанный для генерация данных любого формата, используя фаззинг.

Возможности фаззинг платформы:

- Генерация данных любого формата
- Реализуемость coverage-guided fuzzer-a
- Настройка фаззера путем изменения конфигурации

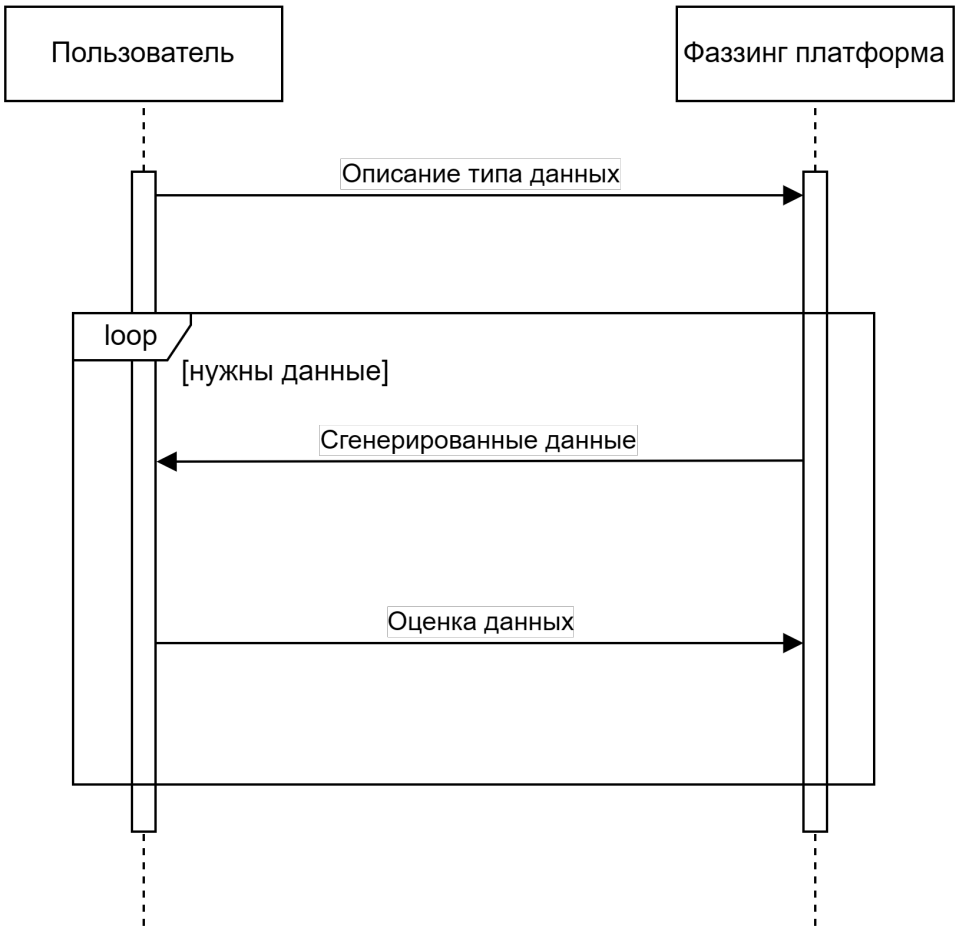

Рис. 1: Схема работы с фаззинг платформой

На рисунке 1 представлена схема работы с фаззинг платформой.

2.3. Выводы

В результате анализа существующих фаззеров для языка Go не было найдено решения, которое бы позволило генерировать значения всех типов, существующих в языке Go, кроме функций. Самой широкой поддержкой типов обладает комбинация go-fuzz и gofuzz, но даже в этом случае нет поддержки каналов и интерфейсов. Также ни один из фаззеров не предоставляет информацию о покрытии кода. В результате было принято решение использовать фаззинг платформу, как основу для построения на ней coverage-guided fuzzer-а.

3. Обзор существующих генераторов модульных тестов для языка Go

Существует 3 наиболее интересных генератора модульных тестов для Go: Symflower[4], NxtUnit[5], FinalUnit[6]. Дальше пойдет речь подробно о каждом из них.

3.1. Symflower

Symflower представлен в трех видах: приложение командной строки, плагин для GoLand и плагин для Visual Studio Code. Он использует символьное исполнение, а в качестве SMT-решателя выступает Z3 [12]. Symflower умеет генерировать тесты, а также готовые шаблоны тестов. Он хорошо справляется со своей задачей, когда дело касается простых функций, но имеет ряд ограничений, из-за чего даже не запускается на функциях посложнее. Рассмотрим эти ограничения.

Во-первых, Symflower не поддерживает множество типов данных языка Go, а именно: нет поддержки комплексных чисел, хеш-таблиц, каналов, функций, интерфейсов, среди пользовательских типов поддерживаются только те, что определены в одном пакете с тестируемой функцией, также не поддерживаются некоторые комбинации типов, как, например, массив слайсов или слайс массивов. Все это накладывает серьезные ограничения на тестируемые функции.

Во-вторых, Symflower не запускается на файле с хотя бы одним внешним импортом, что опять же сильно сужает его применимость.

В-третьих, Symflower не поддерживает вызов стандартных функций внутри тестируемой.

На рисунке 2 представлена функция, для которой Symflower не смог сгенерировать ни одного теста. В этой функции интересным является возможность переполнения значения переменной res, и, хотя Z3 позво-

[4]Symflower: `https://symflower.com/en/`
[5]NxtUnit: `https://github.com/bytedance/nxt_unit`
[6]FinalUnit: `https://github.com/wimspaargaren/final-unit`

```go
func IntSearch(n int) {
    res := n*n - 2*n + 1
    if res < 0 {
        panic( v: "")
    }
}
```

Рис. 2: Пример функции, для которой Symflower не сгенерировал ни одного теста

ляет находить переполнения, в этом примере Symflower с задачей не справился.

На рисунке 3 представлена простая функция, возвращающая ошибку, если пришло число больше нуля, и nil в обратном случае. На рисунке 4 изображен единственный сгенерированный тест. Стоит заметить, что времени Symflower было достаточно и он отработал раньше установленного тайм-аута.

```go
func returnErrorOrNil(n int) error {
    if n > 0 {
        return errors.New( text: "")
    } else {
        return nil
    }
}
```

Рис. 3: Тестируемая функция.

```go
func TestSymflowerReturnErrorOrNil1(t *testing.T) {
    n := 0

    actual := returnErrorOrNil(n)

    expected := error(nil)

    assert.Equal(t, expected, actual)
}
```

Рис. 4: Сгенерированные тесты.

Еще одним ограниченим является возможность запуска генерации тестов только для одной функции, лежащей в файле.

Все эти ограничения и примеры плохой работы сильно сказываются на желании пользоваться данным инструментом.

3.2. NxtUnit

NxtUnit представлен только в виде приложения командной строки и появился в марте 2023 года. Он использует технику случайного тестирования, т.е случайно генерирует входные данные для функций. При покрытии новых участков кода, он сохраняет входные данные и резуль-

тат выполнения функции для того, чтобы в дальнейшем сгенерировать тесты. Запустить генерацию тестов можно для функций и методов, лежащих в одном файле. Он поддерживает большое количество типов данных языка Go. Но из-за случайного выбора входных данных для функций он показывает плохое покрытие даже на простых примерах, что отбивает всякий интерес к его использованию.

3.3. FinalUnit

FinalUnit представлен только в виде приложения командной строки. Можно генерировать тесты для функций и методов. Он поддерживает все типы данных в Go. Но к способам генерации значений многих типов есть ряд вопросов. Так, например, числа генерируются случайно из интервала от -100 до 100, в качестве строк используются предопределенные строки из имен и фамилий, каналы генерируются пустые, в качестве функций генерируются заглушки, где единственной строчкой в теле функции является возврат случайных значений.

Главной задачей FinalUnit ставит максимизацию общего тестового покрытие для созданного набора тестов. Это делается с помощью эволюционного машинного обучения. Идея этого типа машинного обучения очень проста. Мы начинаем с популяции организмов, группы существ с некоторой формой ДНК. Затем эта популяция начинает эволюционировать. У каждого организма есть приспособленность, эта оценка пригодности показывает, насколько хорошо этот организм функционирует в мире. Организмы с высокой приспособленностью имеют больше шансов на размножение. Теория утверждает, что это повысит общую приспособленность популяции с течением времени, ведь самые сильные гены имеют больше шансов оказаться в следующем поколении, тем самым увеличивая общую приспособленность.

Каждый организм в FinalUnit состоит из набора тестов для всех функций в данном каталоге. Приспособленность такого организма определяется тестовым покрытием тестового набора. Развивая эту совокуп-

ность, FinalUnit пытается со временем увеличить общее покрытие и найти набор тестов с максимальным покрытием. Генератор останавливается, когда достигнуто целевое покрытие или заданное количество поколений не показало никаких улучшений.

Когда эволюция завершена, берется организм с наилучшей приспособленностью. Этот тестовый набор выполняется, и выходные данные всех тестируемых функций записываются и записываются в файл в качестве результирующих тестовых случаев. В случае, если выходные данные функции недетерминированы, генератор добавит *FIXME*, сообщающий пользователю о добавлении утверждений вручную.

Помимо примитивной генерации входных данных для функций к недостаткам можно отнести также генерацию большого числа лишних тестов. Это происходит из-за отсутствия реализации получения покрытия на одном из тестов. В итоге FinalUnit генерирует все тесты из выбранного лучшего тестового набора.

4. Архитектура приложения

В этой главе представлено описание этапов работы реализованного генератора модульных тестов для языка Go. На рисунке 5 изображена схема работы приложения.

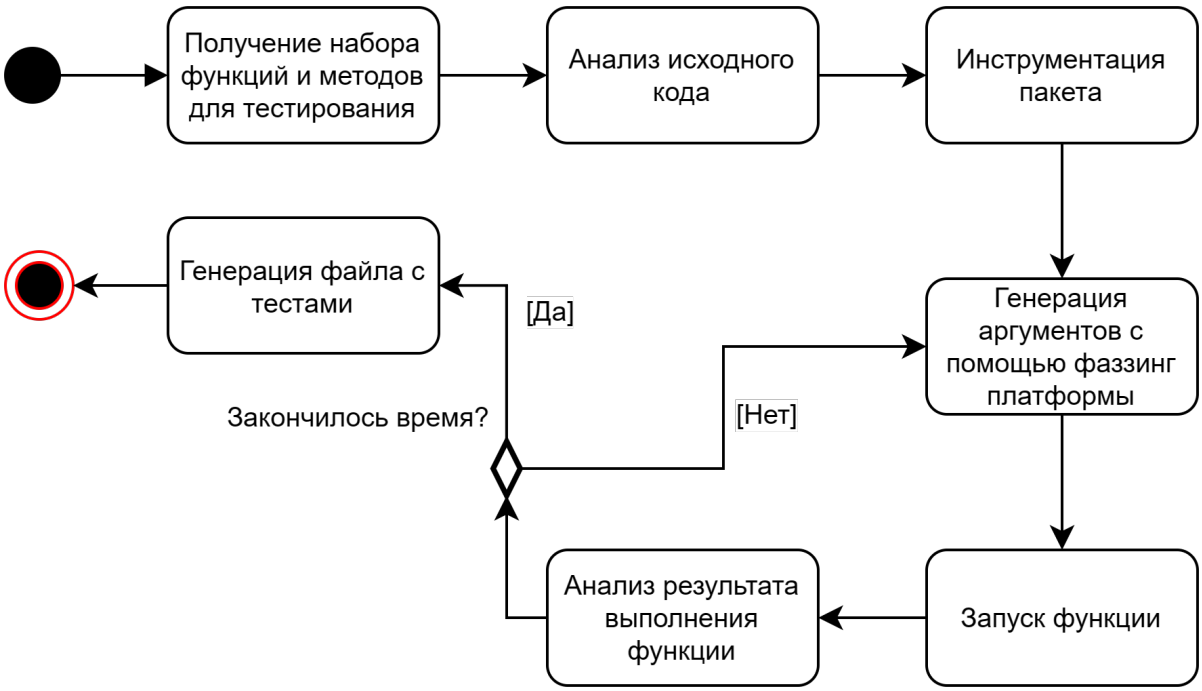

Рис. 5: Схема работы приложения

4.1. Получение набора функций и методов

Первым этапом работы генератора является получение от пользователя необходимой информации для генерации тестов, а именно:

- Пути до файла, содержащего функции и методы, для которых нужно сгенерировать тесты;
- Названий тестируемых функций и методов.

Пользователь может поменять ограничения на время выполнения каждой функции в отдельности и на время общего выполнения, по умолчанию это 1 и 60 секунд соответственно.

Также пользователь может выбрать количество процессов, которые будут заниматься фаззингом, и специальный режим фаззинга - это режим, в котором генерация входных данных заканчивается в момент, когда найден вход, приводящий к ошибке выполнения функции или превышению ограничения на время ее выполнения, при этом в итоге будут сгенерированы тесты только для этих случаев.

4.2. Анализ исходного кода

Вторым этапом является анализ исходного кода. Во время этого этапа происходит поиск выбранных для тестирования функций и методов в коде, сбор информации о типах параметров и результирующих значений, проверка возможности генерации для них тестов: проверка, что функция или метод не является обобщенным, что среди типов в сигнатуре нет типа функций.

Также выполняется извлечение константных значений из тел функций. Константные значения могут помочь в дальнейшем для генерации более качественных входных данных.

4.3. Инструментация пакета

На данном этапе происходит добавление в каждую функцию и метод каждого файла из пакета, в котором лежит тестируемый файл, после каждой строчки логгирования о прохождении данной строчки во время выполнения функции/метода. Логгирование выглядит как увеличение определенного счетчика. Счетчиками являются элементы глобального массива. Номер конкретного счетчика определяется хэш-функцией, отображающей номер инструментированной строки в число от 0 до 65535.

Помимо этого, для условного оператора if:

- Если он не имеет блока else, добавляется пустой блок else. Пример представлен ниже:

```
if x {
}
// Add else because we want coverage for
// "not taken".
if x {
} else {
}
```

- Если он имеет блок else if, то он заменяется на else, где в тело вставляется if. Пример представлен ниже:

```
// The elses are special, because if we have
if x {
} else if y {
}
// we want to cover the "if y". To do this,
// we need a place to drop the counter,
// so we add a hidden block:
if x {
} else {
    if y {
    }
}
```

Также для оператора switch добавляется default блок в случае его отсутствия.

4.4. Генерация аргументов с помощью фаззинг платформы

Следующим этапом является генерация тестовых входных данных для функций/методов с помощью фазизнг платформы. Перед этим создаются файлы для процессов, который будут заниматься запуском функций/методов с переданными им аргументами: файл со счетчиками и файл с логикой запуска функций/методов. Важным является то,

что файл с логикой является тестовым, так как Go, кроме тестов, позволяет запускать только main функцию из main пакета, поэтому генерируется именно тест-функция, которую можно запускать и которая уже может лежать в любом пакете. После создания файлов, происходит компиляция их со всеми зависимостями в бинарный файл, для того чтобы в дальнейшем не тратить время на компиляцию при перезапуске процессов. Далее запускается столько процессов, сколько указал пользователь. Эти процессы дальше будут называться процессами-исполнителями.

Для каждого процесса-исполнителя запускается свой главный процесс со своей фаззинг платформой. Для генерации аргументов фаззинг платформе предоставляются описания нужных типов.

4.5. Запуск функции

Фаззинг платформа генерирует значения, которые отправляются процессу-исполнителю. Передача данных осуществляется с помощью их отправки в виде JSON по сокет соединению. На стороне процесса-исполнителя происходит создание реальных Go объектов с помощью библиотеки reflect. Также с помощью этой библиотеки происходит запуск функции и сериализация результата и пути ее выполнения в JSON, которые в дальнейшем отправляются главному процессу.

4.6. Анализ результата выполнения функции

Существует 4 результата выполнения функции:

- Когда выполнение успешно завершилось
- Когда выполнение завершилось с ошибкой
- Когда выполнение аварийно завершилось
- Когда выполнение превысило ограничение по времени

Для каждого набора покрытых строк сохраняется по одному результату с самыми на каждый возможный сценарий, т.е. максимум может быть 4 результата выполнения функции для одного набора покрытых строк.

4.7. Генерация файла с тестами

Последним этапом является генерация файла с тестами. На этом этапе для каждого тест-кейса генерируется тест в зависимости от того, какой был результат выполнения функции.

Есть 4 варианта тестов:

- Когда выполнение функции успешно завершилось. Пример на рисунке 6

```
func TestLenOfListByUtGoFuzzer(t *testing.T) {
    l := (*List)(nil)

    actualVal := LenOfList(l)

    expectedVal := 0

    assert.Equal(t, expectedVal, actualVal)
}
```

Рис. 6: Пример теста, информирующего об успешном завершении выполнении функции

- Когда выполнение функции завершилось с ошибкой. Пример на рисунке 7
- Когда выполнение функции аварийно завершилось. Пример на рисунке 8
- Когда был превышено ограничение на время выполнение функции. Пример на рисунке 9

```
func TestReturnErrorWithNonNilErrorByUtGoFuzzer(t *testing.T) {
    actualErr := ReturnError()

    expectedErrorMessage := "Error!"

    assert.ErrorContains(t, actualErr, expectedErrorMessage)
}
```

Рис. 7: Пример теста, информирующего об завершении выполнении функции с ошибкой

```
func TestDivOrPanicPanicsByUtGoFuzzer(t *testing.T) {
    x := 0
    y := 0

    expectedVal := "div by 0"

    assert.PanicsWithValue(t, expectedVal, func() {
        _ = DivOrPanic(x, y)
    })
}
```

Рис. 8: Пример теста, информирующего об аварийном завершении выполнении функции

```
// SumOfChanElements((<-chan int)(nil)) exceeded 1000 ms timeout
```

Рис. 9: Пример теста, информирующего о превышении ограничения на время выполнения функции

4.8. Ограничения

Ниже перечислены ограничения, которые есть у моего приложения:

- Тесты генерируются для функций, лежащих в одном файле
- Нет поддержки обобщенных функций
- Не поддерживается тип функций
- Нет гарантии корректной работы для функций, зависящих от глобального состояния
- Нет гарантии корректной работы для многопоточных функций

5. Сравнение с аналогами

Результатом предыдущих частей работы является генератор модульных тестов для языка Go, который имеет название UnitTestBot Go. В данной главе приведено краткое сравнение получившегося решения с аналогами.

5.1. Сравнения с существующими решениями

Для сравнения с аналогами были выбраны функции из репозитория TheAlgorithms/Go[7]. Для начала было принято решение сравниться главным образом с Symflower, который использует символьное исполнение. Для этого были выбраны простые функции, на которых Symflower смог запуститься. Такие функции не обладали большой вложенностью, вызовами стандартных функций и использованием других пакетов. В итоге было выбрано 38 функций, на которых и производились сравнения.

	Всего функций	Функций, для которых сгенерировались тесты	Количество сгенерированных тестов	Покрытие, %
UnitTestBot Go	38	38	149	98.3
Symflower	38	37	133	96.6
FinalUnit	38	38	732	89.8
NxtUnit	38	35	35	54.0

Рис. 10: Сравнение с аналогами на простых функциях

На рисунке 10 изображены результаты сравнения. Как видно из таблицы, UnitTestBot Go показал лучший результат по покрытию и отработал на всех функциях. Symflower не смог отработать на одной из функций и по итогу немного отстал по покрытию. Особое внимание стоит уделить результату работы FinalUnit, потому что было сгенери-

[7]Репозиторий с алгоритмами, написанными на языке Go: `https://github.com/TheAlgorithms/Go`

ровано 732 теста при нелучшем показателе покрытия. На рисунке 11 изображен пример 2-ух из 18 тестов, сгенерированных им для функции, определяющей является ли матрица единичной (рисунок 16). Все 18 тестов проходят одинаковые пути исполнения функции и при этом результат выполнения функции также совпадает, что говорит о практически полной бесполезности 17 из них. UnitTestBot Go же сгенерировал только 4 теста (рисунок 17) и все они отличаются покрытыми путями исполнения, при этом покрытие составило 100%.

```
func (s *MatrixisidentitySuite) TestIsIdentity0() {
    matrix := [3][3]int{[3]int{5, 92, -78}, [3]int{11, 13, -57}, [3]int{-72, -9, 2}}
    out := IsIdentity(matrix)
    s.False(out)
    _ = out
}

func (s *MatrixisidentitySuite) TestIsIdentity1() {
    matrix := [3][3]int{[3]int{-57, -18, 34}, [3]int{57, 95, 47}, [3]int{-8, 64, -99}}
    out := IsIdentity(matrix)
    s.False(out)
    _ = out
}
```

Рис. 11: Пример 2-ух из 18 сгенерированных тестов FinalUnit для функции, определяющей является ли матрица единичной (рисунок 16)

Дальше были выбраны сложные функции из того же репозитория. Эти функции уже были больше, обладали большой вложенностью и сложными условиями. В итоге было выбрано 21 функция.

	Всего функций	Функций, для которых сгенерировались тесты	Количество сгенерированных тестов	Покрытие, %
UnitTestBot Go	21	21	211	94.9
FinalUnit	21	21	391	66.3
NxtUnit	21	21	21	52.0
Symflower	21	5	23	13.9

Рис. 12: Сравнение с аналогами на сложных функциях

На рисунке 12 изображены результаты сравнения. Как видно из таблицы, опять же UnitTestBot Go показал лучший результат по покрытию и отработал на всех функциях. Аналоги же сильно отстали по покрытию. Symflower смог отработать только на 5 функциях и показал худшее общее покрытие. На рисунках 13 и 14 изображено покрытие UnitTestBot Go и FinalUnit на весьма простой функции, ищущей строку ABC (зеленым отмечены покрытые участки кода, красным - нет). Единственная сложность в этой функции заключается в существовании трех вложенных условных операторов, с которой FinalUnit справиться не смог.

```
3  func StringSearch(str string) bool {
4      if len(str) != 3 {
5          return false
6      }
7      if str[0] == 'A' {
8          if str[1] == 'B' {
9              if str[2] == 'C' {
10                 return true
11             }
12         }
13     }
14     return false
15 }
```

Рис. 13: Покрытие UnitTestBot Go на функции, ищущей строку ABC.

```
3  func StringSearch(str string) bool {
4      if len(str) != 3 {
5          return false
6      }
7      if str[0] == 'A' {
8          if str[1] == 'B' {
9              if str[2] == 'C' {
10                 return true
11             }
12         }
13     }
14     return false
15 }
```

Рис. 14: Покрытие FinalUnit на функции, ищущей строку ABC.

5.2. Выводы

Основываясь на вышеизложенном, можно сделать следующие выводы:

- Реализованный генератор показывает лучше покрытие тестами, чем существующие аналоги
- Количество сгенерированных тестов меньше или сравнимо с аналогами

Заключение

Главным результатом данной работы стало создание генератора модульных тестов для языка Go с использованием фаззинга. В качестве фаззера был реализован coverage-guided fuzzer на основе фаззинг платформы. Использование фаззинга для генерации тестовых входных данных позволило избежать проблем символьного исполнения, а также позволило добиться лучших показателей в покрытии кода по сравнению с существующими аналогами.

Генератор поддерживает генерацию тестов для функций и методов, а также поддерживает все типы данных Go, кроме функций.

В итоге генератор был интегрирован в проект UnitTestBot для языка Java:

- Создано приложение командной строки для демонстрации работы на веб-сайте utbot.org
- Плагины для IntelliJ IDEA Ultimate и GoLand интегрированы в плагин UnitTestBot для языка Java

Направлениями дальнейшей работы могут быть:

- Поддержка генерации тестов для обобщенных функций
- Минимизация набора генерируемых тестов

Список литературы

[1] Aizatsky M., Serebryany K., Chang O., Arya A., and Whittaker M. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. — 2016. — Access mode: https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[2] Manès V. J. M., Han H., Han C., Cha S. K., Egele M., Schwartz E. J., and Woo M. The Art, Science, and Engineering of Fuzzing: A Survey. — 2018. — Access mode: http://export.arxiv.org/pdf/1812.00140.

[3] Barrett C., Kroening D., and Melham T. Problem Solving for the 21st Century: Efficient Solvers for Satisfiability Modulo Theories. — 2014. — Access mode: https://theory.stanford.edu/~barrett/pubs/BKM14.pdf.

[4] CERT. CERT database of security vulnerabilities. — 2017. — Access mode: https://kb.cert.org/vuls/.

[5] Candea G. and Godefroid P. Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances. — 2019.

[6] Liang H., Pei X., Jia X., Shen W., and Zhang J. Fuzzing: State of the Art. — 2018.

[7] Howard M. and Lipner S. The Security Development Lifecycle. — Microsoft Press, 2006. — Access mode: https://archive.org/details/securitydevelopm0000howa/page/n9/mode/2up.

[8] LibFuzzer – a library for coverage-guided fuzz testing. — Access mode: https://llvm.org/docs/LibFuzzer.html.

[9] Microsoft. Project OneFuzz. — 2020. — Access mode: https://www.microsoft.com/en-us/research/project/project-onefuzz/.

[10] Roy Osherove. The Art of Unit Testing. — Manning Publications, 2009. — ISBN: 1933988274.

[11] Wikipedia. Evolutionary algorithm. — Access mode: https://en.wikipedia.org/wiki/Evolutionary_algorithm.

[12] Wikipedia. Z3 Theorem Prover. — Access mode: https://en.wikipedia.org/wiki/Z3_Theorem_Prover.

[13] Zalewski Michał. AFL. — 2013. — Access mode: https://lcamtuf.coredump.cx/afl/.

Приложение

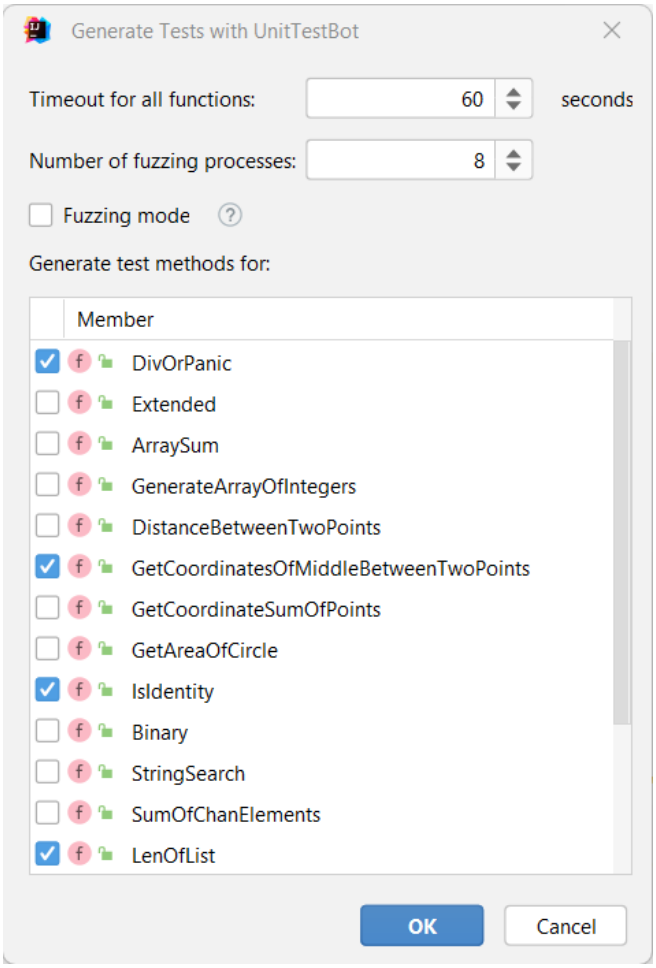

Рис. 15: Интерфейса плагина

```
func IsIdentity(matrix [3][3]int) bool {  18 usages
    for i := 0; i < 3; i++ {
        for j := 0; j < 3; j++ {
            if i == j && matrix[i][j] != 1 {
                return false
            }

            if i != j && matrix[i][j] != 0 {
                return false
            }
        }
    }
    return true
}
```

Рис. 16: Функция, определяющая является ли матрица единичной

```go
func TestIsIdentityByUtGoFuzzer1(t *testing.T) {
    matrix := [3][3]int{{1, 0, 3}, {0, 0, 0}, {0, 0, 0}}

    actualVal := IsIdentity(matrix)

    assert.False(t, actualVal)
}

run test | debug test
func TestIsIdentityByUtGoFuzzer2(t *testing.T) {
    matrix := [3][3]int{{0, 0, 0}, {0, 0, 0}, {0, 0, 0}}

    actualVal := IsIdentity(matrix)

    assert.False(t, actualVal)
}

run test | debug test
func TestIsIdentityByUtGoFuzzer3(t *testing.T) {
    matrix := [3][3]int{{1, 0, 0}, {0, 0, 0}, {0, 0, 0}}

    actualVal := IsIdentity(matrix)

    assert.False(t, actualVal)
}

run test | debug test
func TestIsIdentityByUtGoFuzzer4(t *testing.T) {
    matrix := [3][3]int{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}

    actualVal := IsIdentity(matrix)

    assert.True(t, actualVal)
}
```

Рис. 17: Пример сгенерированных тестов для функции, определяющей является ли матрица единичной (рисунок 16)