Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б15-мм

Автоматическая генерация тестов для веб приложений на платформе ASP.NET

Алексеев Артур Игоревич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры СП, к. ф.-м. н., Д. А. Мордвинов

Консультант:
инженер-исследователь, М. П. Костицын

Санкт-Петербург
2023

Оглавление

Введение

Тестирование приложений является неотъемлемой частью процесса разработки, которая позволяет выявить недостатки и ошибки, а так же улучшить надежность и безопасность продукта. Однако, писать тесты — это большая и трудная задача, которая может отнять много времени у программиста. Также, из-за человеческой ошибки, не все ситуации в программе легко предугадать, и при тестировании приложения можно не увидеть важные ошибки и уязвимости. Символьное исполнение представляет собой технику анализа кода приложения, которая позволяет найти все наборы входных данных для выполнения всех возможных путей. Эта техника автоматизирует процесс создания тестов, а так же гарантирует, что ни один случай не будет пропущен.

С появлением все большего числа веб-приложений и ростом их роли в нашей повседневной жизни, необходимость в их тестировании становится все более актуальной. Помимо улучшения опыта использования, появится возможность выявлять критические уязвимости приложений, которые могут использоваться злоумышленниками. В контексте веб-приложений особенно важно интеграционное тестирование, которое позволяет проверить работу приложения в целом, включая взаимодействие различных компонентов и систем.

Для создания веб-приложений можно использовать различные инструменты, в том числе платформу .NET[1] и ASP.NET Core[3]. Они предоставляют широкие возможности для разработки и тестирования приложений, а также обладают высокой производительностью и масштабируемостью. ASP.NET Core также имеет открытый исходный код, и является кроссплатформенным.

Для автоматической генерации тестов в .NET существует инструмент V♯[6]. V♯— это движок символьного исполнения для программ на платформе .NET. Он использует символьное исполнение для создания тестов, которые покрывают все возможные пути выполнения приложения. С помощью V♯уже можно генерировать модульные тесты для веб-приложений, но исследовать приложение полностью, от получения

запроса до его обработки и возвращения ответа пользователю, не представляется возможным. Наша задача - расширить возможности символьного движка V♯и исследовать с его помощью веб-приложения на основе ASP.NET Core.

Дата сборки: 26 декабря 2023 г.

1. Постановка задачи

Целью данной работы является автоматическая генерация тестов для веб приложений на платформе ASP.NET Core с помощью V♯. Для достижения цели были сформулированы следующие задачи.

1. Изучить литературу, посвященную символьному исполнению.
2. Создать алгоритм для генерации тестов.
3. Спроектировать архитектуру решения.
4. Реализовать архитектуру в проекте.
5. Провести тестирование.

2. Обзор

В данной главе описывается общий алгоритм работы механизма символьного исполнения, приводятся некоторые примеры приложений на ASP.NET, а так же рассматривается написание интеграционных тестов с использованием WebApplicationFactory.

2.1. Символьное исполнение

Символьное исполнение представляет собой технику анализа программного обеспечения, которая позволяет найти все наборы входных данных, способствующие выполнению каждого из его возможных путей. Вместо того чтобы запускать программу на конкретных входных данных, символьное исполнение работает с символьными значениями, которые представляют переменные и память в программе.

Процесс символьного исполнения начинается с создания символьных значений для всех входных данных программы. Затем анализатор программы строит граф потока управления, который показывает все возможные пути выполнения программы. Для каждого пути выполнения анализатор выполняет символьное вычисление значений переменных на каждом шаге выполнения программы.

Важным понятием в символьном исполнении является символьное состояние, которое отражает состояние программы в конкретной точке исполнения. Частями символьного состояния являются

1. Инструкция — инструкция, которая исполняется в данный момент

2. Ограничения условия пути π — множество ограничений на символьные значения переменных, при которых поток исполнения дойдет до данной точки. Например, если до этого встречалась конструкция if(a $>$ 10), и мы вышли в ветку then, тогда в ограничении условия пути будет условие a $>$ 10.

3. Символьная память — отображение из множества объектов программы в множество их символьных значений.

Символьная машина выполняет инструкции программы, обновляя инструкцию, ограничения условия пути и память у состояний. В случае, когда исполнение программы может пойти не одним путем, создаются новые состояния, с соответствующими значениями ограничения условия пути.

После того как символьное исполнение завершено для всех возможных путей выполнения программы, анализатор может использовать полученную информацию для выявления потенциальных ошибок и уязвимостей, или для создания тестов с большим уровнем покрытия. Например, если в результате работы символьного движка получился ряд состояний, можно подобрать параметры, подходящие под ограничения условия пути, и использовать их как входные данные для тестов, которые покроют все возможные случаи.

2.2. Пример работы символьного движка

Листинг 1: Функция target

```
bool target(int x, int y){
x = 4;
if (x > y * 2) return true;
if (x < y) return false;
error();
}
```

Для исследования символьным движком функции target, создается начальное состояние, в котором инструкция — первая инструкция в функции, а ограничение условия пути — true. Далее, создается и меняется ряд состояний

1. При исполнении первой инструкции, x = 4, в символьной памяти переменная x принимает значение 4

2. После этого исполняется инструкция if(x > y * 2), которая порождает 2 состояния.

3. Одно находится в ветке then, с ограничением пути, удовлетворяющем условию x > y * 2, $\pi = 4 > \alpha_y * 2$ и на инструкции return true, после исполнения которой результат функции target будет true

4. Другое состояние находится в ветке else, на инструкции if (x < y), с ограничением пути !(x > y * 2), то есть $\pi = !(4 > \alpha_y * 2)$. При исполнении данной инструкции появится еще 2 состояния

5. Одно находится в ветке then, с ограничением пути, удовлетворяющем условию x > y * 2 && x < y, $\pi = !(4 > \alpha_y * 2)$ && $4 < \alpha_y$ и на инструкции return false, после исполнения которой результат функции target будет false

6. Другое находится в ветке else, с ограничением пути $\pi = !(4 > \alpha_y * 2)$ && $!(4 < \alpha_y)$, на инструкции error(), после исполнения которой результатом функции target будет ошибка

На основе трех получившихся финальных состояний (3, 5, 6), можно сделать вывод о поведении функции при различных входных данных, и создать тесты. Но для этого нужно узнать конкретные значения переменных, которые удовлетворяют формуле, заданной в ограничениях условия пути. Задача поиска значений символьных переменных, при которых условие пути выполняется, может быть представлена как задача выполнимости в формулы в теориях, и для решения этой задачи используются SMT решатели. SMT решатели могут найти конкретные значения символьных переменных для заданного условия, например, для 3 состояния решением будет x = 0, y = 1, для второго — x = 0, y = 5, а для состояния 6 — x = 0, y = 3. Схема символьного исполнения представлена на Рис. 1.

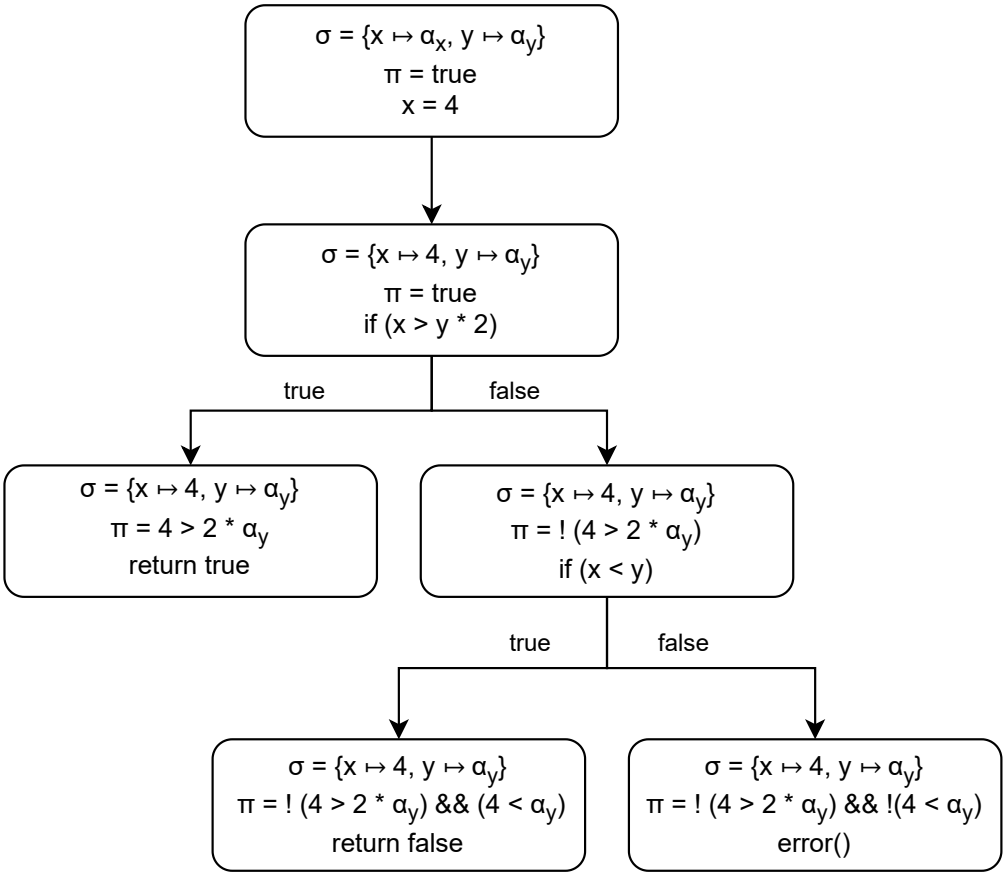

Рис. 1: Схема символьного исполнения для функции target

2.3. Веб-программирование на платформе .NET

.NET[1] — это платформа для разработки приложений, которая поддерживает несколько языков программирования, включая C#, F# и VB.NET. Она предоставляет множество инструментов для разработки приложений, включая среду разработки Visual Studio, библиотеки классов .NET и множество других инструментов. Она также поддерживает множество платформ, включая Windows, Linux и macOS.

Для создания веб-приложений в .NET имеется ряд фреймворков, таких, как

1. ASP.NET Core
2. Coalesce
3. FubuMVC

4. NancyFx

5. IISNode

6. Giraffe

ASP.NET Core[3] — это фреймворк для веб-разработки, который основан на платформе .NET, разработанный компанией Microsoft и предоставляет множество инструментов и библиотек для разработки веб-приложений, включая поддержку MVC (Model-View-Controller), Web API, SignalR и многое другое. Он также предоставляет возможность разработки кроссплатформенных приложений, которые могут работать на Windows, Linux и macOS. Согласно исследованию[2], проведенному сайтом Stack Overflow в 2023 году, ASP.NET Core был назван самым популярным веб-фреймворком среди разработчиков на платформе .NET.

2.4. Структура проектов на основе ASP.NET Core

ASP.NET Core является гибким фреймворком, который позволяет создавать проекты с разной структурой и архитектурой. Например

1. MVC (Model-View-Controller) — это структура проекта, которая предполагает наличия моделей (предоставляет данные и реагирует на команды контроллера, изменяя своё состояние), представление (отвечает за отображение данных модели пользователю, реагируя на изменения модели) и контроллер (интерпретирует действия пользователя, оповещая модель о необходимости изменений).

2. Web API — это структура проекта, которая предназначена для создания веб-сервисов и API. Она не содержит представлений и предоставляет только контроллеры и модели, которые могут возвращать данные в формате JSON или XML.

3. Razor Pages — это структура проекта, которая предоставляет более простой подход к созданию веб-страниц. Она использует Razor-

шаблоны для отображения данных и не требует контроллеров для обработки запросов.

4. Blazor — это структура проекта, которая позволяет создавать SPA-приложения с помощью C# и .NET. Она использует компонентную модель и может работать как на сервере, так и в браузере.

5. Minimal API — это структура проекта, которая позволяет писать минимум кода, отказываясь от многих паттернов и конструкций.

Особое место в данных архитектурах занимает именно MVC. На этой модели основаны многие примеры, она очень удобна для создания больших приложений и хорошо масштабируется. Структура MVC (Model-View-Controller) в ASP.NET Core представляет собой следующие элементы

1. Models — это классы, которые описывают данные и бизнес-логику приложения. Они могут содержать методы для работы с данными, валидацию и другие функции.

2. Views — это файлы, которые отображают данные пользователю. Они могут быть написаны на Razor-шаблонах или на других языках, таких как HTML, CSS и JavaScript.

3. Controllers — это классы, которые обрабатывают запросы пользователя и взаимодействуют с моделями и представлениями. Они могут содержать методы для получения данных из моделей, передачи данных в представления и другие функции.

4. Startup.cs — это файл конфигурации приложения, который содержит настройки сервисов, маршрутизации и другие параметры.

5. wwwroot — это папка, которая содержит статические файлы, такие как изображения, CSS и JavaScript.

Структура проекта MVC в ASP.NET Core может быть изменена в зависимости от требований приложения, но эти основные элементы остаются неизменными.

Рассмотрим структуру MVC на примере немного видоизмененного стартового проекта, который можно создать, выполнив команду dotnet new mvc.

При создании приложения будет создан ряд директорий, описанных выше, а конкретно

1. Controllers, содержащая контроллер HomeController для отображение домашней страницы и страницы безопасности

2. Models, содержащая модель данных ошибки, ErrorViewModel

3. Views, содержащая общие страницы разметки и представления для контроллера HomeController в соответствующей поддиректории Home (по названию контроллера)

4. wwwroot, содержащая минимальные стили для страниц

А так же файл Program.cs, в описана логика по созданию и запуску приложений. При создании приложения WebApplicationBuilder, с помощью него настраиваются сервисы, далее создается приложение, в котором настраивается конвеер обработки запроса, и затем приложение запускается

Листинг 2: Код файла Program.cs

```
// Созданиебилдера
var builder = WebApplication.CreateBuilder(args);

// Настройкасервисов
builder.Services.AddControllersWithViews();

// Созданиеприложения
var app = builder.Build();

// Настройкаконвеераобработкизапроса
app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();
app.UseAuthorization();

app.MapControllerRoute(
name: ''default'',
pattern: ''{controller=Home}/{action=Index}/{id?}'');

// Запускприложения
app.Run();
```

2.5. Сервер, иссползуемый в ASP.NET Core

По умолчанию, сервер, на котором запускается приложение — Kestrel[4]. Согласно документации Microsoft[4], Kestrel — это кроссплатформенный веб-сервер для ASP.NET Core. Kestrel — это рекомендуемый сервер для ASP.NET Core, и он настроен по умолчанию в шаблонах проектов ASP.NET Core. Преймущества Kestrel, согласно документации Microsoft:

1. Кроссплатформенность: Kestrel — кроссплатформенный веб-сервер, работающий в Windows, Linux и macOS.

2. Высокая производительность: Kestrel оптимизирован для эффективной обработки большого количества одновременных подключений.

3. Простота использования: Kestrel оптимизирован для работы в средах с ограниченными ресурсами, например контейнерами и пограничными устройствами.

4. Безопасность: Kestrel поддерживает протокол HTTPS и защищен от уязвимостей веб-сервера.

5. Поддержка расширенных протоколов: Kestrel поддерживает общие веб-протоколы, в том числе: HTTP/1.1, HTTP/2 и HTTP/3, WebSockets

6. Гибкость рабочих нагрузок: Kestrel поддерживает множество рабочих нагрузок:

   (a) ASP.NET приложения, такие как Minimal API, MVC, Razor pages, BlazorSignalR и gRPC.

   (b) Создание обратного прокси-сервера с помощью YARP.

7. Расширяемость: настройка Kestrel с помощью конфигурации и ПО промежуточного слоя. Диагностика производительности: Kestrel предоставляет встроенные функции диагностики производительности, такие как ведение журнала и метрики.

Также, на платформе Windows, присутствуют HTTP.sys и IIS, а так же есть возможность написать свою реализацию сервера, воспользовавшись интерфейсом IServer.

2.6. Интеграционные тесты в ASP.NET Core

Интеграционное тестирование, в отличие от модульного тестирования, позволяет полностью протестировать работу системы. В случае веб-приложения это означает, что тестируется все система, начиная от принятия запроса, заканчивая его обработкой и возвращением результата. Интеграционные тесты гарантируют, что компоненты приложения работают правильно на уровне, включающем поддерживающую инфраструктуру приложения, такую как база данных, файловая система и сеть. ASP.NET Core поддерживает интеграционные тесты с использованием платформы модульного тестирования с тестовым веб-узлом и тестовым сервером в памяти.

На ASP.NET интеграционное тестирование приложения делается с помощью WebApplicationFactory[5]. WebApplicationFactory предоставляет пользователю запустить приложение с сервером TestServer, которое будет работать в памяти. У этого сервера можно создавать клиентов, и моделировать взаимодействие с веб-приложением. Тесты, сделанные с помощью WebApplicationFactory, имеет схожую структуру.

1. Перед самим тестом, создается экземпляр WebApplicationFactory, в котором выполняется подстановка сервера, и запускается фактическое приложение в отдельном потоке, готовое обрабатывать пользовательские запросы. Так же, опционально, может проводиться дополнительная онфигурация, например мокирование некоторых объектов.

2. Сначала, в самом тесте, создается клиент. Он уже подключен к серверу, и любой запрос от него пойдет прямо в приложение.

3. Далее, с помощью метода, позволяющего совершать запросы, делается ряд запросов к серверу, при котором получаются данные, выполняются некоторые эффекты, например запись в базу или создание файла.

4. После, полученные ответы и эффекты проверяются на соответствие требованиям тестировщика

Листинг 3: Пример теста на простой API

```
private WebApplicationFactory<Program> _factory;

[SetUp]
public void Setup()
{
    // Созданиеобъекта  WebApplicationFactory
    _factory = new WebApplicationFactory<Program>();
}

[Test]
public async Task Test()
{
    // Созданиеклиента
    var client = _factory.CreateClient();

    // Взаимодействиессервером
```

```
    var response = await client.GetAsync("api/get");

    // Проверкарезультата
    response.EnsureSuccessStatusCode(); // Status
       Code 200-299
}
```

Список литературы

[1] Microsoft. — .NET. — URL: https://dotnet.microsoft.com/en-us/ (дата обращения: 2023-12-23).

[2] StackOverflow. — Stack overflow developer survey. — URL: https://survey.stackoverflow.co/2023/ (дата обращения: 2023-12-23).

[3] Microsoft. — Документация ASP.NET Core. — URL: https://learn.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[4] Microsoft. — Документация Microsoft. Сервер Kestrel. — URL: https://learn.microsoft.com/ru-ru/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[5] Microsoft. — Интеграционное тистирование ASP.NET Core. — URL: https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-8.0 (дата обращения: 2023-12-23).

[6] VSharp-team. — Репозиторий V. — URL: https://github.com/VSharp-team/VSharp (дата обращения: 2023-12-23).