Метод генерации End-to-end автоматических тестов из естественно-языковых тестовых сценариев на основе предобученной OpenIE модели

К. С. Кобышев[1], С. А. Молодяков[2]
Санкт-Петербургский политехнический университет им. Петра Великого
Высшая школа программной инженерии
[1]kobyshev2.ks@edu.spbstu.ru, [2]molodyakov_sa@spbstu.ru

***Аннотация*. В настоящее время распространена практика покрытия программных продуктов автоматическими тестами. При разработке автоматических тестов отдельно разрабатывается каркас и тесты, в которых вызываются функции каркаса. Предлагается метод генерации E2E-тестов из функциональной спецификации. Он включает следующие основные этапы: формирование тестового сценария из спецификации; разбиение тестового сценария на предложения, которые транслируются в одну строку итогового кода; предложения преобразуется в синтаксическое дерево при помощи предобученной модели OpenIE; при помощи модели Word2Vec наименования тестовых шагов сопоставляются с функциями тестирования; новое семантическое дерево преобразуется в код на языке Kotlin. Особенностью метода является применение синтаксического дерева для генерации тестов и каркаса тестового фреймворка. Описан прототип системы, автоматически генерирующей тесты на языке Kotlin из спецификации на естественном языке.**

Ключевые слова: автоматический тест, обработка естественного языка, кластеризация, E2E-автотест, word2vec, Kotlin.

I. Введение

В настоящее время распространена практика покрытия программных продуктов автоматическими тестами. Покрытие программного обеспечения осуществляется на различных уровнях в рамках пирамиды тестирования: модульные тесты, интеграционные тесты, API (Application programming interface) автотесты, E2E (End-to-End) автотесты [1]. Покрытие программы автотестами также позволяет облегчить трудоемкость процесса рефакторинга кода, а также может служить первичной документацией кода для разработчиков в рамках методологии Test Driven Development [2].

Framework – подход, который позволяет оптимизировать процесс разработки распространенных в промышленных системах API и E2E-автотестов [3]. При разработке автоматических тестов отдельно разрабатывается каркас (framework), состоящий из тестовых шагов и проверок, и тесты, в которых вызываются функции каркаса.

II. Проблемы существующих подходов автоматизации тестирования

В условиях сложного промышленного проекта аналитиками формируется документ, описывающий поведение системы – функциональная спецификация. В случае, если проект является долгоживущим нередко функционал поставляется короткими итерациями (релизными циклами), либо итерации отсутствуют, и выдача сборки на боевое окружение происходит непосредственно после реализации функционала. В этом случае важно проверить не только новый функционал, но и существующий ранее, необходимо провести автоматизированное регрессионное тестирование. Рассмотрим, представленные в табл. I существующие подходы по автоматизации тестирования, и определим их недостатки.

ТАБЛИЦА I Подходы автоматизации тестирования

Хар-ки/подход	Классический	BDD	Методы верификации	Обучение нейросети
Структурированность тестов	-	++	++	--
Участие аналитика	--	++	++	--
Автоматизация составления тестов	--	--	++	+
Стойкость к цикломатической сложности	++	++	--	++
Надежность применяемого метода	+	+	++	--

Существует классический подход по автоматизации тестирования, согласно которому аналитик составляет функциональную спецификацию, по которой QA-инженеры (Quality Assurance engineer) составляют автоматические тесты полностью вручную. Такой подход заставляет аналитиков и QA-инженеров работать отдельно, участие аналитика минимально, взаимодействие аналитика и тестировщика происходит через документ – функциональную спецификацию. Также на QA-инженеров полностью возлагается ответственность за поддержку структуры тестового фреймворка. При таком подходе полностью отсутствует автоматизация составления тестов.

Подход BDD (Behaviour-Driven Development) предполагает составление аналитиком каркаса для тестового фреймворка при помощи предметно-ориентированного языка [4]. Аналитик составляет структуру тестового фреймворка в то время, как QA-инженеры должны реализовать тестовый фреймворк. Такой подход позволяет достичь максимальной структурированности тестов. Тем не менее, данный подход, как и классический подход, полностью исключает автоматизацию составления тестов.

В табл. I в отдельную группу объединены алгоритмы формальной верификации программ. Эти алгоритмы полностью проверяют корректность программы исходя из требований функциональной спецификации, составленных, например, на языке темпоральной логики [5]. Производительность процесса верификации программ сильно замедляется при увеличении цикломатической сложности программы. Процесс формальной верификации предполагает проверку всех возможных состояний программы из-за чего после определенного порога происходит «комбинаторный взрыв состояний программы». Из-за этого формальная верификация, как правило, применима не для исходной программы, а только для ее прототипа.

Также существует подход, основанный на обучении нейронной сети [6]. Авторы предлагают обучить нейронную сеть на поданных в программу случайных входных данных и полученных из нее выходных данных. Данный подход полностью исключает участие аналитика, а тестирование происходит на основе уже построенной программы. Тем не менее, данный подход не может гарантировать надежность ввиду того, что невозможно гарантировать полностью корректное обучение модели нейронной сети. Также невозможно надежно продолжить обучение модели нейронной сети в соответствии с новыми изменениями программы.

Таким образом, были выявлены следующие проблемы существующих подходов автоматизации тестирования:

- Хаотичность, отсутствие структуры тестов.
- Работа аналитиков в отдельности от QA-инженеров, отсутствие правильного понимания у QA-инженеров об ожидаемом поведении программы.
- Процесс составления автоматических тестов проводится полностью вручную и требует значительных трудовых ресурсов.
- Резкое падение производительности системы автоматических тестов при увеличении цикломатической сложности тестируемой программы.
- Отсутствие гарантии того, что система автоматического тестирования построена правильно.

В данном исследовании был разработан алгоритм, который позволяет устранить перечисленные проблемы.

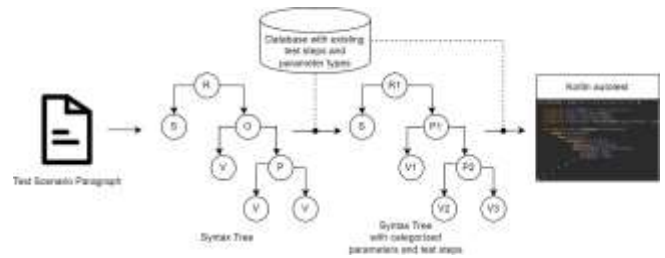

Рис. 1. Предлагаемое решение для автоматической генерации тестов

III. Автоматизация разработки тестов

Рассмотрим решение, предлагаемое в данном исследовании, схематично отображенное на рис. 1. Предлагается выстроить процесс разработки автоматических тестов следующим образом:

- Аналитик составляет функциональную спецификацию системы в виде набора тестовых сценариев на естественном языке.
- Тестовые сценарии на естественном языке преобразуются в код автотестов и в интерфейсы, которые представляют собой тестовые шаги.
- QA-специалист программирует реализацию интерфейсов тестовых шагов.

Рассмотрим подробнее предлагаемую программную систему.

IV. Шаги метода генерации автоматических тестов

Рассмотрим на верхнем уровне, как работает предлагаемый алгоритм для генерации автоматических тестов (схематично отображено на рис. 2). Алгоритм выполняет следующие шаги:

1. Алгоритм берет название сценария и главы документации и называет соответствующим именем тестовый метод и класс автотеста.
2. Тестовый сценарий разбивается на предложения. Каждое предложение алгоритм разбирает по отдельности и транслирует в одну строку итогового кода.

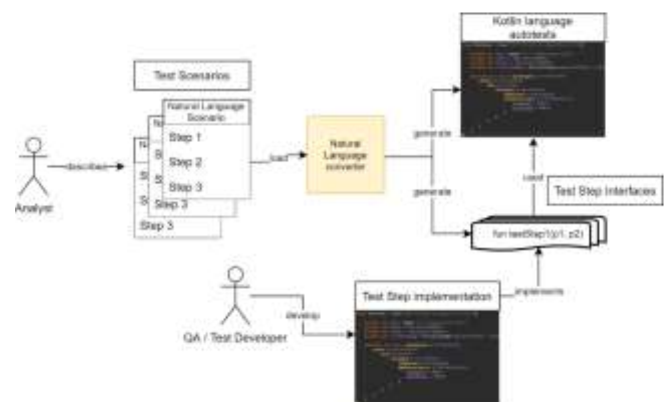

Рис. 2. Шаги предлагаемого алгоритма

3. Каждое предложение преобразуется в синтаксическое дерево при помощи предобученной модели OpenIE [7].

4. При помощи предобученной модели Word2Vec [8, 9] наименования тестовых шагов, групп параметров и параметров сопоставляются с имеющимися объектами в базе данных.

5. Полученное новое семантическое дерево преобразуется в код на языке Kotlin.

Рассмотрим подробнее 3, 4 и 5 шаги алгоритма.

V. Построение синтаксического дерева

В предлагаемом решении используется алгоритм OpenIE [7] для построения синтаксического дерева из пункта тестового сценария. Предлагаемый авторами алгоритм сначала подготавливает текстовые данные: токенизация [10], лемматизация [11], определение частей речи составляющих предложение слов [12], построение дерева зависимостей слов предложения *D* [13]. При помощи предобученной модели формируются триплеты по выражению (1), где *s* – это субъект, *R* – это отношение, *o* – это объект:

$$T_i = s_i R_i o_i \tag{1}$$

Объект может представлять собой набор нескольких взаимосвязанных слов естественного языка. В этом случае объект возможно представить в виде части от дерева зависимостей, то есть согласно выражению (2):

$$o_i \in D_i \tag{2}$$

Такое представление позволяет представить объект, как иерархическую структуру из множества различных параметров, что сделает авто тест более наглядным.

Дерево зависимостей объекта возможно представить в виде выражений (3) и (4), где P – узлы дерева, и V – листья, значения. При этом параметры P могут состоять из других параметров P, либо из значений V, то есть *o* возможно представить в виде иерархической структуры:

$$o_i = P \cup V = (P_1, P_2, \dots, P_k) \cup (V_1, V_2, \dots, V_k) \tag{3}$$

$$\forall n, P_n = (P_x, P_{x+1}, \dots, P_y) \cup (V_m, V_{m+1}, \dots, V_l) \tag{4}$$

По результату данного этапа найденные субъекты, отношения, наборы параметров и значения пока что не соотнесены ни с каким из типов. На следующем этапе происходит их классификация для того, чтобы сформировать каркас тестового фреймворка.

VI. Определение типов полученных элементов теста

В результате выполнения предыдущего этапа из тестовых сценариев, составленных на естественном языке, получены иерархически связанные между собой субъекты *s*, отношения *R*, наборы параметров *P*, значения *V*. Каждый из *s, R, P, V* соотнесен с каким-либо исходным словом или набором слов естественного языка. Любое слово естественного языка можно представить в виде вектора координат в семантическом пространстве. Близкие по координатам *s, R, P, V* можно сгруппировать в кластеры для того, чтобы затем эти кластеры представить в виде методов интерфейсов, составляющих тестовый фреймворк.

На данный момент существует множество различных способов получения координат слов естественного языка в семантическом пространстве. Наиболее часто используемые на сегодняшний день: RNNLM [14], word2vec [8], GloVe [15], fastText [16]. В предлагаемой реализации алгоритма генерации авто тестов предполагается использование модели GloVe, которая в значительной степени учитывает частоту совместной встречаемости слов при вычислении их взаимных координат.

Как было сказано, на предыдущем этапе алгоритма были получены синтаксические деревья и множество (s, R, P, V). Также перед тем, как начать кластеризацию, мы уже имеем множество (s^{c0}, R^{c0}, P^{c0}, V^{c0}), сопоставленное со множеством кластеров (s_0\`, $R_0$\`, P_0\`, $V_0$\`), которые были найдены ранее во время кластеризации слов из предыдущих предложений тестового сценария.

Каждое из подмножеств s, R, P, V делится на кластеры отдельно. Возьмем в качестве примера Рис. 3. Рассмотрим случай, когда уже из предыдущих предложений тестового сценария были определены кластеры s_1^c, s_2^c. Допустим, осталось еще соотнести с кластерами 3 предложения тестового сценария, из которых получены субъекты s_3, s_4, s_5. У каждого из них по 4 координаты в семантическом пространстве, а на рисунке показана двухмерная проекция. Кластеры определяются следующим образом. Мы получаем точку. Если в радиусе r от точки нет ни одного кластера, то в центре данной точки размещается кластер, который ассоциируется с данной точкой. Если точка попадает в зону другого кластера, то она входит в данный кластер. Если точка не вошла в кластер, но ее окружность радиусом r пересекается с каким-либо из кластеров, то данная точка не образует новый кластер, а входит в ближайший кластер.

На рис. 3 видно, что изначально были найдены кластеры s_1^c, s_2^c. Затем на алгоритм была подана точка s_3, которая была определена в кластер s_3^c. Окружность точки s_4 пересекается с кластером s_3^c, поэтому она была определена в кластер s_3^c. Точка s_5 была определена внутри изначально имеющегося кластера s_1^c.

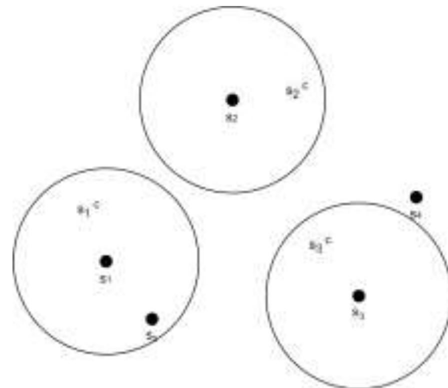

Рис. 3. Пример кластеризации на двухмерной проекции семантического пространства

После данного этапа остается последний этап – получение кода на языке Kotlin.

VII. Преобразование семантического дерева в код на языке Kotlin

Последним этапом необходимо преобразовать полученное типизированное синтаксическое дерево в код на языке Kotlin. В итоге получается автотест на

предметно-ориентированном языке и каркас тестового фреймворка в виде интерфейсов. Рассмотрим на табл. II, каким образом осуществляются преобразования.

ТАБЛИЦА II ПРЕОБРАЗОВАНИЯ СЕМАНТИЧЕСКОГО ДЕРЕВА

Тип преобразования	До	После

Субъект	User paid free package
	User - subject
		user {
		…paid free package...
		}

Склеивание субъектов	User paid free package. User got payment bill.	user {
		...paid free package, got payment bill…
		}

Отношение	User paid free package	user {
		paid(…)
		}

Объект	User paid free package	user {
		paid(Package(…))
		}

Параметр	User paid free package	user {
		paid(Package(type=…)
		}

Значение	User paid free package	user {
		paid(Package(type=FREE)
		}

Тестовый сценарий	Payment flow:
	User paid free package. User got payment bill.
		@Test
		fun paymentFlow() {
		user {
		paid(Package(type=FREE)
		got(PaymentBill())
		}
		}

Найденный субъект преобразуется в лямбда-выражение с контекстом. QA-инженеру предлагается реализовать класс контекста. Если один и тут же субъект встречается в двух пунктах тестового сценария, то происходит склеивание двух лямбда-выражений в одно. Найденное отношение соответствует вызову метода, который так же предлагается реализовать. Параметр представляет собой название поля внутри объекта или другого параметра. Значение может встретиться внутри параметра. В итоге весь код оборачивается в тестовый метод, название которого совпадает с названием тестового сценария.

VIII. ЗАКЛЮЧЕНИЕ

В результате данного исследования был проработан алгоритм для частичной автоматизации разработки тестов, который позволяет обеспечить высокую структурированность системы тестирования, обеспечить единое понимание ожидаемого поведения программного продукта у аналитиков и инженеров по обеспечению качества, достигнуть высокой надежности системы тестирования и высокой стойкости к цикломатической сложности программного продукта. Из тестовых сценариев на естественном языке формируются автоматические тесты, а также интерфейсы на языке Kotlin, которые предлагается реализовать инженерам, поддерживающим тестовую систему. В дальнейшем планируется протестировать разработанный алгоритм на предмет точности, скорости, полноты покрытия программного продукта тестами.

СПИСОК ЛИТЕРАТУРЫ

[1] Radziwill N., Freeman Gr. Reframing the Test Pyramid for Digitally Transformed Organizations // Software Quality Professional. 2020. V. 22. №4. P. 18-25.

[2] Karac It., Turhan B. What Do We (Really) Know about Test-Driven Development? // IEEE Software. 2018. V. 35. №4. P. 81-85. DOI: 10.1109/MS.2018.2801554

[3] Fountoura M. A. Systematic Approach for Framework Development. Rio de Janeiro, 1999. 165 p.

[4] Irshad M., Britto R., Petersen K. Adapting Behavior Driven Development (BDD) for large-scale software systems // Journal of Systems and Software. 2021. V. 177. 110944. DOI: 10.1016/j.jss.2021.110944

[5] Wasira W. Existing Tools for Formal Verification and Formal Methods // MS Computer Science, Lewis University. 2020. DOI: 10.13140/RG.2.2.12162.22721.

[6] Данилов А.Д, Мугатина В.М. Верификация и тестирование сложных программных продуктов на основе нейросетевых моделей // Вестник ВГТУ. 2016. Т. 12. №6. С. 62-67.

[7] Angeli G., Premkumar M., Manning Chr. Leveraging Linguistic Structure For Open Domain Information Extraction // Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, 2015. №1. P. 344-354. DOI: 10.3115/v1/P15-1034.

[8] Long M., Yanqing Z. Using Word2Vec to process big text data // IEEE International Conference. 2015. №10. DOI: 10.1109/BigData.2015.7364114.

[9] Ковалев А.Д., Никифоров И.В., Дробинцев П.Д. Автоматизированный подход к семантическому поиску по программной документации на основе алгоритма Doc2Vec // Информационно-управляющие системы. 2021. № 1 (110). С. 17-27.

[10] Garcia-Teruel R. M., Simon-Moreno H. The digital tokenization of property rights. A comparative perspective. // Computer Law & Security Review. 2021. V. 41. №2. P. 1-16. DOI:10.1016/j.clsr.2021.105543.

[11] Vimala B., Lloyd-Yemoh E. Stemming and Lemmatization: A Comparison of Retrieval Performances. // Lecture Notes on Software Engineering. 2014. №2. P. 262-267. DOI: 10.7763/LNSE.2014.V2.134.

[12] Chotirat S., Meesad P. Part-of-Speech tagging enhancement to natural language processing for Thai wh-question classification with deep learning // Heliyon. 2020. V. 7. №10. DOI: 10.1016/j.heliyon.2021.e08216.

[13] Zmigrod R., Vieira T., Cotterell R. On Finding the K-best Non-projective Dependency Trees // ACL/IJCNLP. 2021. DOI:10.18653/v1/2021.acl-long.106.

[14] Lecorve G., Motlicek P. Conversion of Recurrent Neural Network Language Models to Weighted Finite State Transducers for Automatic Speech Recognition // 13th Annual Conference of the International Speech Communication Association 2012.

[15] Pennington J., Socher R., Manning Chr. Glove: Global Vectors for Word Representation. // EMNLP. 2014. №14. P. 1532-1543. DOI: 10.3115/v1/D14-1162.

[16] Khasanah I. N. Sentiment Classification Using fastText Embedding and Deep Learning Model // Procedia Computer Science. 2021. №189. P. 343-350. 10.1016/j.procs.2021.05.103.