An algorithm of test generation from functional specification using Open IE model and clustering

K. S. Kobyshev[1], S. A. Molodyakov[2]
Peter the Great Saint-Petersburg Polytechnic university
High School of Programming Engineering
[1]kobyshev2.ks@edu.spbstu.ru, [2]molodyakov_sa@spbstu.ru

***Annotation*. The practice of automatic test covering is widespread now. Usually, framework and tests are developed separately, and framework functions are used in tests. We proposed an algorithm to generate E2E tests from functional specification. The algorithm includes the following main steps: test scenarios forming from specification; test scenarios splitting to sentences that will be translated to the one final code line; sentences transformation to syntax tree using pretrained OpenIE model; test steps comparison with testing functions using Word2Vec model; given semantic tree transformation to the Kotlin language code. The algorithm feature is an application of syntax tree to generate tests and framework interfaces. The paper contains the description of protype of system automatically generating Kotlin language tests from natural language specification.**

Key words: automatic test, natural language processing, clustering, E2E test, word2vec, Kotlin.

I. INTRODUCTION

The practice of automatic test is widespread now. The covering can be implemented on different levels of testing pyramid: unit tests, integration tests, API (Application programming interface) tests, E2E (End-to-End) tests [1]. The program autotest covering lets to decrease complexity of code refactoring process, also tests can be used as primary code documentation according to Test-Driven Development methodology [2].

Framework is an approach that allows to optimize the development process of API and E2E tests, that are used in enterprise systems often [3]. Usually, framework and tests are separately developed, and framework functions are used in tests.

II. PROBLEMS OF EXISTING TESTING AUTOMATION SOLUTIONS

When programming system is quite complex, analysts prepare a document describing system behavior called functional specification. Usually, in case of complex and long living project, the functionality should be delivered by short iterations (release cycles) or build should be delivered immediately after functionality implementation. In this case it is necessary to check not only the new functionality, but also existing earlier, necessary to complete the automated regression testing. Consider the testing automation methods presented in Table I and define their disadvantages.

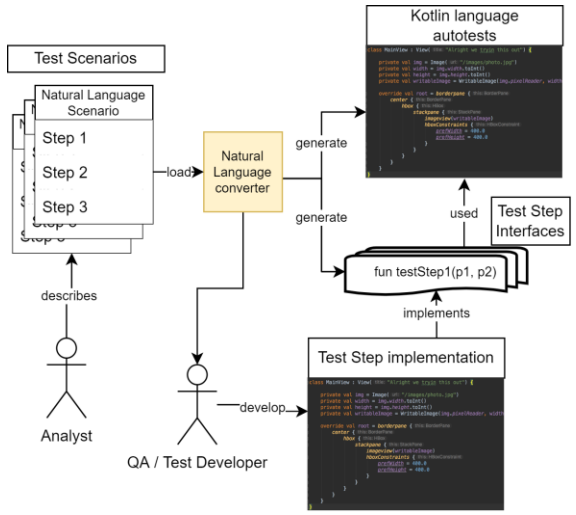
TABLE I. TESING AUTOMATION METHODS

Ch-cs/approach	*Classic*	*BDD*	*Verification methods*	*Neural network training*
Test structuredness	-	++	++	--
Analyst partipiation	--	++	++	--
Test configuration automation	--	--	++	+
Cyclomatic complexity resistance	++	++	--	++
Reliability of applied method	+	+	++	--

According to the classic testing automation method, analyst prepare the functional specification that is used for automatic test preparation by QA engineers (Quality Assurance engineer). Automatic tests are prepared manually. This method forces analyst and QA engineers to work separately. Participation of analysts is minimal and interaction between analysts and QA engineers is done over the document – functional specification. Also, QA engineers are responsible of test framework structure support. This approach excludes the full automation of test preparation.

The BDD approach (Behavior-Driven Development) is a test framework interfaces preparation by analyst with using of domain-oriented language [4]. Analysts prepare structure of test framework and QA engineers implements the test framework. This approach allows to achieve the best test structuredness. Unfortunately, this approach like classic approach, excludes the full automation of test preparation.

Algorithms of formal verification methods are collected to the one group in the Table I. These algorithms allow to completely check the program correctness according to functional specification requirements, made with, for example, language of temporal logic [5]. The performance of verification process significantly degrades with increasing of cyclomatic complexity of program. The formal verification process is a check of all possible program states, which can cause the “combinatorial explosion”. Therefore, the formal verification usually applied for prototype of program instead of the source program.



Pic. 1. The proposed solution for automatic test generation

Also, there is an approach based on the training of neural network [6]. Authors proposed to train neural network by random input data for program and given from its output data. This approach does not take in account analyst participation and testing is based on already prepared program. But this approach cannot guarantee the reliability because it is impossible to make the completely correct trained neural network model. Also, it is impossible to continue the model training with new program changes.

So, the following problems were found out during the existing methods analysis:

- Chaotic state, absence of test structure.
- Analysts work separately from QA engineers, absence of correct unified understanding of expected system behavior. Their work can be done only through documents, functional specification, consisted of non-strict natural language sentences.
- Automated test configuring is done in manual mode and requires significant labor resources.
- Low testing system performance with increasing of cyclomatic program complexity.
- Absence of guarantee that automatic testing system is completely correct.

An algorithm allowing to avoid enumerated disadvantages was proposed in the research.

III. Test Development Automation

Consider the solution proposed in the current research and schematically presented in Pic. 1. We proposed to build the development process in the following way:

- Analysts prepare functional specification in a form of natural language scenario set.
- Natural language test scenarios are transformed to the autotest code and interfaces of test steps by the proposed automatic software tool.
- QA engineer implements given test step interfaces on Kotlin programming language.

Consider the proposed solution in detail.

IV. Test Generation Algorithm Steps

Consider the work of proposed test generation algorithm on high level (schematically presented on Pic. 2). The proposed method includes the following steps:

1. The functional specification chapter is taken as a test class name, and test scenario name is taken as a test method name.
2. The test scenario is divided to sentences. Each sentence will be transformed to the one line of final code.
3. Each sentence is transformed to the syntax tree using the pretrained OpenIE model [7].
4. Test step, parameter group and separate parameter names are associated with test step, parameter group and parameter types using Word2Vec model [8, 9].
5. The given semantic tree is transformed to the Kotlin language code.

Consider steps 3, 4, 5 in detail.

V. Algorithm of Syntax Tree Preparation

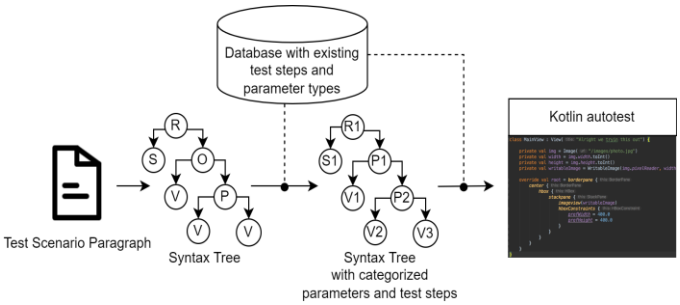
OpenIE model is used to build the syntax tree from test scenario sentence [7]. Before OpenIE processing, the text data should be prepared by the following algorithms: tokenization [10], lemmatization [11], part-of-speech definition [12], building the dependency tree D [13]. Triplets are formed with using of OpenIE according to the expression (1), where s is a subject, R is a relation, o is an object:

$$T_i = s_i R_i o_i \quad (1)$$

In some cases, an object contains a set of several interconnected natural language words. The object can be presented in a form of a part of dependency tree, therefore according to the expression (2):

$$o_i \in D_i \quad (2)$$

This view allows to present the object as a hierarchic



Pic. 2. Steps of the proposed algorithm

structure of different parameters, that will make automatic tests more descriptive. The dependency tree can be presented by expressions (3) and (4), where P are tree nodes, and V are leaves. In other words, these leaves are values V of parameters P. And parameters P can include other parameters P or values V, so *o* can be presented in a form of hierarchic structure, so tests will contain trees of parameters P with values V:

$$o_i = P \cup V = (P_1, P_2, .., P_k) \cup (V_1, V_2, .., V_k) \quad (3)$$

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

For now, when the current step is done, found subjects, relationships, parameter sets, and values are not associated with any types. In the next step, they will be classified to form interfaces of test framework.

VI. Test Element Type Definition

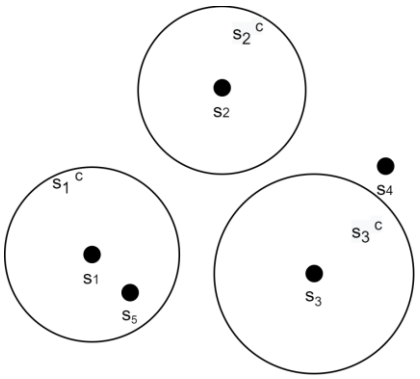
As a result of the previous step, we got a hierarchically connected subjects *s*, relationships *R*, parameter sets *P*, values *V*. Each *s, R, P, V* is associated with some source natural language word or word set. Any natural language word can be presented in a form of coordinates vector in semantic space. Close *s, R, P, V* can be grouped to clusters associated with test framework interfaces.

For now, there are many ways to get natural language word coordinates in semantic space. The most used for today models presenting word semantic coordinates are: RNNLM [14], word2vec [8], GloVe [15], fastText [16]. The GloVe model was used in the proposed method because this model takes in account in significant degree word cooccurrence frequency, that is important for our clustering.

As it was discussed earlier, we got a syntax tree D and a set (s, R, P, V). Also, before clustering, we have a set (s^{c0}, R^{c0}, P^{c0}, V^{c0}), associated with a cluster set (s_0`, $R_0$`, P_0`, $V_0$`) found earlier on clustering of previous test scenario sentence words.

Each subset s, R, P, V is divided to clusters separately. Consider an example in the Pic. 3 in two-dimensional space, when clusters s_1^c, s_2^c already found from previous test scenario sentences and for now we want to parse 3 remaining sentences and define their s, R, P, V types or clusters.

After parsing of three remaining sentences, as a result, algorithm extracts subjects s_3, s_4, s_5 from these three sentences. Clusters of these subjects are defined in the following way. So, we get a point in the two-dimensional semantic space. If there



Pic. 3. Clusterization on two-dimensional projection of semantic space

are no clusters in radius r from the given point, then the cluster with radius r will be placed at this point and the point will be a cluster center. If the point is in the other cluster zone, then this point will be associated with that cluster. If the point is not in cluster, but the r-radius circle from this point intersects with any cluster, then the point will be associated with the closest cluster.

We can see on the Pic. 3 that clusters s_1^c, s_2^c were found at the beginning. Then algorithm accepted the point s_3, that was associated with the cluster s_3^c, because the r-radius circle from this point is not intersected with any existing r-radius clusters. The r-radius circle of point s_4 is intersected with cluster s_3^c, that is why it was associated with the cluster s_3^c. The point s_5 was associated with the cluster s_1^c because it was inside of the r-radius circle of this cluster.

The last remaining step is to get the Kotlin language code from the given semantic tree.

VII. Semantic Tree Transformation to the Kotlin Language Code

The last step is to get the Kotlin language code from the given typed semantic tree. As a result, we will get an autotest on the domain-oriented language and interfaces of the test framework. Consider transformation rules presented in the Table II, where you can see examples of the parsed sentence in the "before" column and prepared automatic test code fragment in the "after" column.

TABLE II. Semantic Tree Transformations

Transformation rule	***Before***	***After***

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

Subject grouping	User paid free package. User got payment bill.	user {
		...paid free package, got payment bill…
		}

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

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

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

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

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

The found subject is transformed to the lambda expression with context. QA engineer should implement the context class. If the same subject is appeared in two test scenario sentences, then those subject lambda expressions will be grouped to the one lambda expression. The found relationship is transformed

to the method call, and that method should be implemented. Parameters are transformed to the class field names. Values are transformed to the primitive types of the Kotlin language or Strings. Then all code is wrapped to the test method having the name like the test scenario name.

VIII. Prototype of the proposed solution

A prototype of the proposed solution was implemented on Java language. The developed system uses pretrained OpenIE model in a form of Maven package manager dependency called Stanford NLP. A pretrained GloVe model was used. This model was given from Wikipedia of 2014 year and Gigaword text corpuses. The model contains 400 thousand words and their coordinates in 100-dimensional space and takes 822 Mb of memory. The GloVe model was stored and indexed in Mongo database. For now, the prototype gives true results for simple test scenarios, however, we found that it does not work correctly in some complex test scenarios including multiple words in subjects and relationships. Therefore, we need to investigate more and improve clustering stage of the proposed algorithm for now.

IX. Conclusion

We proposed the algorithm for automatization of test development that allows to provide high test structuredness, the unified understanding of system behavior of analysts and QA engineers, to achieve the high reliability and resistance to cyclomatic complexity of test system. Automatic tests and test framework interfaces on Kotlin language are formed from natural language test scenarios, and QA engineers should implement interfaces of test framework. In the future, we want to test accuracy, speed, recall of the developed algorithm, also we want to improve the clustering stage of the proposed algorithm.

References

[1] N. Radziwill “Freeman Gr. “Reframing the Test Pyramid for Digitally Transformed Organizations”. *Software Quality Professional*. 2020. vol. 22, №4. pp. 18-25.

[2] It. Karac and B. Turhan. “What Do We (Really) Know about Test-Driven Development?” *IEEE Software*. 2018. vol. 35. №4. pp. 81-85. DOI:10.1109/MS.2018.2801554

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

[4] M. Irshad, R. Britto and K. Petersen. “Adapting Behavior Driven Development (BDD) for large-scale software systems”. *Journal of Systems and Software*. 2021. №17. 20 p. DOI: 10.1016/j.jss.2021.110944

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

[6] A. D. Danilov and V. M. Mugatina. “Verification and testing of complex software products based on neural network models”. *Vestnik VGTU* [VSTU Bulletin]. 2016. vol. 12. №6. pp. 62-67. (in Russian)

[7] G. Angeli, M. Premkumar and Chr. Manning. “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. Bejing, 2015. №1. pp. 344-354. DOI: 10.3115/v1/P15-1034.

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

[9] A. D. Kovalev, I. V. Nikiforov and P. D. Drobincev. “Automated approach for semantic search in software documentation based on Doc2Vec algorithm”. *Informaciono-upravlayuscshiye sistemi* [Information control systems]. 2021. №1 (110). pp. 17-27. (in Russian)

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

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

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

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

[14] G. Lecorve and P. Motlicek. “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] J. Pennington, R. Socher and Chr. Manning. “Glove: Global Vectors for Word Representation”. EMNLP. 2014. №14. pp. 1532-1543. DOI: 10.3115/v1/D14-1162.

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