International Journal for Housing Science and Its Applications
Publish August 6, 2025. Volume 46, Issue 3 Pages 7257-7266

https://doi.org/10.70517/ijhsa463621

Research and application of RESTful API interface testing technology based on simulation environment construction

Fuju Sun[1,*]
[1] School of Information Technology and Intelligent Manufacturing, Shanghai Xingjian College, Shanghai, 200072, China
Corresponding authors: (e-mail: Shamysun@163.com).

Abstract The popularization of microservice architecture and cloud computing has driven RESTful API interfaces to become the core vehicle for service interaction. In this paper, we propose a test case generation algorithm PSST based on solution space tree to improve the quality of RESTful API interfaces. Combining OpenAPI specification and combinatorial testing theory, we construct a hierarchical nested API architecture by analyzing the path, parameter and response mapping relationship of the interface definition file. The PSST algorithm for generating combinatorial test cases based on the solution space tree is introduced, and the case generation process is optimized based on the greedy algorithm. The results show that the reward value of PSST algorithm is 9.524. 100% state coverage and key module coverage are realized when 22 and 50 test cases are generated. The test program runs with no more than 40% CPU and memory usage and fast IO read/write growth. The RESTful API interface after the use case generation test is able to satisfy the access requirements of 750 users at the same time.

Index Terms RESTful API interface, solution space tree, PSST algorithm, greedy algorithm, test case generation

I. Introduction

Nowadays, with the development and growth of the Internet, one software product after another comes out. The Internet and software continue to change people's perceptions and lifestyles, but also improve the quality of life, at the same time, people have higher requirements for the function, performance and practicality of software [1], [2]. Enterprises are faced with increasing business needs, and software systems have gradually become complex, how to ensure the quality of software products has become a great concern for major software companies.

As integration between software systems becomes more and more common, many applications and services rely on external application program interfaces (APIs) to obtain data, perform operations or realize some specific functions [3], [4]. API interface testing is an important part of the software project development process, and occupies an increasing proportion in the software life cycle [5]. The purpose of a testing system is to simulate a user situation as realistically as possible in an actual test environment [6]. The implementation of API interface testing is used to ensure that the application or system can run correctly and stably, and provide a good experience to the user [7], [8]. By conducting integrated and comprehensive API interface testing, some potential failures and risks of an application can be effectively reduced, and the quality and reliability of the application can be improved [9]-[11]. However, traditional manual testing can no longer meet the needs of software testing, and the importance of automated testing that simulates the user's environment is becoming more and more prominent [12], [13]. Among them, RESTful API interface effectively improves the application program interface fuzzy test effectiveness and test efficiency through automatic generation of use cases, pre-screening and other methods, which is of great significance to improve the security of application product interface [14]-[16].

Based on the layered design principle of REST style, this paper divides the API system into resource layer, logic layer and front-end layer, and clarifies the functions and interactions of each layer. Aiming at the low efficiency of traditional fuzzy testing, we propose a parsing method based on the interface definition file to extract key information such as paths, methods, parameter vectors, etc., and construct a dynamic mapping model of parameters and responses. The solution space tree model is further introduced to describe the solution space structure of multi-parameter combinations. Design the PSST algorithm based on the solution space tree, traverse the solution space tree by backtracking, and then use the greedy complement algorithm to generate test cases one by one, to realize full coverage of the solution set and improve the effect of RESTful API interface testing.

II. Restful API interface structure and testing technology analysis

II. A. Restful API interface analysis

II. A. 1) REST style

Representation Layer State Transition (REST) is a system architecture style for Internet applications that utilizes Uniform Resource Identifiers (URIs) to locate resources and HTTP verbs to describe operations.REST consists of three main elements: resource, representation, and state. Resource is a physical object or abstract concept embodied as a stream of bits in the network world, and each resource is identified by a unique URI, usually represented by a URL. Representation refers to a certain form of resources presented for the construction of loosely coupled, scalable Web applications to provide standards, the operation of the resource is by first obtaining the existing representation of the resource, and then complete the internal transformation from the existing state to the target state, the operation of the resource that is the HTTP verb POST, DELETE, PUT, GET corresponding to the addition, deletion, modification and check of the four kinds. The state can be both the server-side resource state and the end-application state, the state of the resource is saved on the server side, and the application state is maintained by the application itself.

REST separates the different concerns of the client-server model's client-side user interface and server-side data storage, simplifying the server part and improving its scalability and portability across different platforms. All interactions are stateless, each invocation request from the client to the server carries all the information needed by the server to process the request, and no state information is retained or memorized on the server side. Unified standard interaction interface, simplify the architecture, improve the visibility of the interaction between the business logic, the separation of business logic and standardized interface, so that the client and the server are extended separately without interfering with each other. REST commonly used simple, lightweight, easy to use, readable JSON data format as a unified interface standard for interaction.

With the development of mobile Internet and cloud service applications, REST is used by more and more developers for its simple design, easy to read, lightweight packet transmission and other features, and is gradually used to replace the complex and bulky SOAP Web Service model.

II. A. 2) Restful API structure

An API can transform several resources in a resource portfolio from one state to another, Figure 1 shows the internal structure of an API, which consists of three parts: access control, API definition and business logic, the access control specifies the degree of openness of the API, the permission requirements and the monitoring of the performance and quality, etc., and the API definition specifies the function of the API, the access method, the URL, the parameter constraints and the return results, etc., while the business logic is the main body that completes the whole API function and realizes the business logic. The API definition part specifies the functional role of the API, access methods, URL, parameter constraints, return results, etc., while the business logic completes the whole API function and realizes the main body of the business logic, which is a combination of resources that returns the API function by interacting with resources such as databases, business systems, the Internet or local APIs and carrying out the appropriate logic processing.

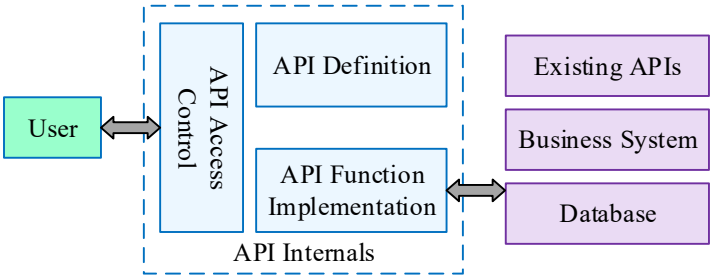

Figure 1: API Structure

In the Internet environment, the public API interface only needs to provide the client with detailed information about the resources and other related information, without considering the use and business functions of the client to obtain the resources. However, as an enterprise or campus internal API design, not only should there be an interface to obtain resources, but more importantly, it is necessary to consider how the resources are utilized, and how the interface granularity size can be designed to be convenient for the client to use and other internal business, so as to allow the business system to easily complete the business logic without excessive processing of the interface data.

According to the internal business needs and RestfulAPI design principles, the API architecture of the internal environment is reconstructed into a multi-layer nested API architecture. From the inside out, the following are in order: resource APIs that are oriented to independent resource data, have the smallest granularity, and contain almost no business, logical APIs that need to call resource APIs to handle related logical business or functional requirements, and front-end APIs that are directly oriented to the client users and need to provide personalized resources to each terminal. Client APIs can be evolved to correspond to the layers of MVC.

Figure 2 shows the three-tier API architecture. The three-layer API structure with nested relationships is divided according to the function and form of the interface, not a logical structural layering. The layers are connected to each other, the lower layer provides services and output resources for the upper layer, the upper layer obtains resources from the lower layer to complete its own mission function, and externally, the interface of each layer exists as an independent entity, which is open to the client and provides interface services.

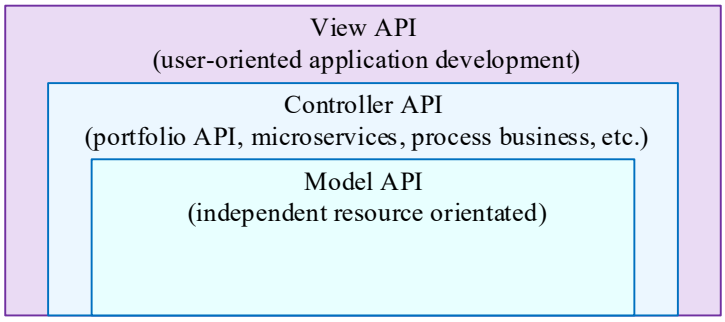

Figure 2: Three-tier API architecture

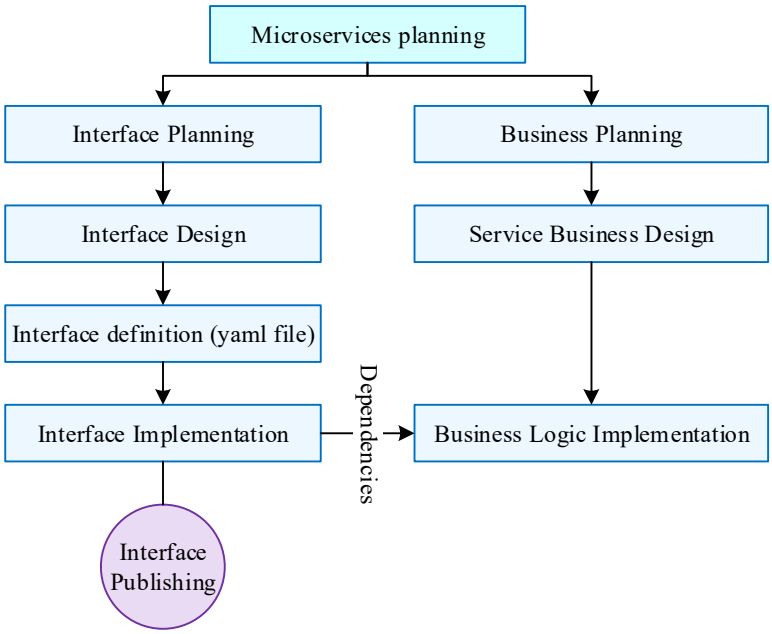

Figure 3: Microservice interface release process

II. B.Interface Definition Based Test Case Generation Approach

II. B. 1) RESTful Interface Definition File

For interface testing, how to construct interface test case messages is especially critical. Fuzzy testing is usually realized by reverse message parsing and message splicing. Take the PeachFuzz tool as an example, its interface fuzzy test case generation method is to generate fuzzy messages by obtaining message request messages to speculate the attack mode. Most of the fuzzy messages generated by this method will be directly filtered by the interface checksum and cannot reach the program. The efficiency of such fuzz testing is relatively low.

In the OpenAPI specification, the definition of a pickup OpenAPI interface is itself a JSON object, which can be represented as a JSON file or a YAML file. In the OpenStack standard, the interface for service publishing is a

RESTful interface that conforms to the OpenAPI 3.0 specification, and Figure 3 shows an implementation flow of the microservice interface. This flow shows that in this system, the RESTful interface is defined by a YAML file that conforms to the OpenAPI3.0 specification.

YAML is a superset of JSON, and its advantage is that it can easily support annotations, line break splitting, multi-line strings, etc., and it can also reference files. So the JSON format parameters in the message body can be defined by YAML, and YAML can also be used externally by way of annotations to declare the parameter's data type and boundary values, interface SLAs, and other information. Since JSON and YAML ultimately define the same content, the representation is similar, only in the syntax format differences.

II. B. 2) Interface Definition File Parsing and Use Case Structure Generation

The interface definition file is essentially a full-measurement representation of the state information of the RESTful API interface, so the essence of parsing the interface definition file is to extract the key descriptive information of the RESTful API interface in the definition.

The key information of RESTful API interface includes path, method, request message body, and response. We make the following abstraction of the interface definition model:

For an interface I of a characteristic, given a path U, method M, the expected response R_0 obtained by this interface is only related to the parameter vector $K_{[\,]}$ in the message body of the telegram in a RESTful request and the value vector $V_{[]}$ it is passed. To wit:

$$R_0 = I\{U, M, (K_0, K_1 \cdots K_n)^T (V_{00}, V_{10} \cdots V_{n0})\} \tag{1}$$

Therefore, test case structure generation for a single interface is really about determining the correspondence between the dynamic information in the request message and the response of a RESTfulAPI interface request message while keeping the request message statically unchanged. That is, the correspondence between the message body parameter vector and the response vector. That is:

$$R_x \leftrightarrow (K_0, K_1 \cdots K_n)^T (V_{0x}, V_{1x} \cdots V_{nx}) \tag{2}$$

Meanwhile, for each specific interface, the parameter vectors $K_{[\,]}$ contained in the interface, and the mapping relationship between K and V are relatively fixed, so the correspondence between the expected result of the test case and the body of the test message is in fact the relationship between the response message and the value vector:

$$R_x \leftrightarrow V_{0x}, V_{1x} \cdots V_{nx} \tag{3}$$

$$(R_0, R_1, R_2 \cdots R_m)^T \leftrightarrow \begin{bmatrix} V_{00}, V_{10} \cdots V_{n0} \\ V_{01}, V_{11} \cdots V_{n1} \\ \vdots \\ V_{0m}, V_{1m} \cdots V_{nm} \end{bmatrix} \tag{4}$$

II. C. Test case generation method based on solution space tree

II. C. 1) Solving spatial tree models

A solution space tree is a tree that represents the solution structure of a problem in terms of a tree structure based on the properties of the problem to be solved, and represents the solution of the problem in terms of a path from the root node to the leaf nodes. Let there be m input parameters affecting the SUT: $P = \{p_1, p_2 \cdots p_m\}$, there are n different cases of values for each parameter P_i, and $V_i = \{v_1, v_2 \cdots v_n\}$ denotes the specific cases of values for parameter P_i.

1) Completely permutations and combinations of the value cases of all the parameters of the SUT are performed, and then the result is mapped into a tree of height $m+1$: the first level of the tree root node has V_1 branches representing the value case V_1 of parameter P_1, the second level of the tree root node each has V_2 branches representing the value case V_2 of parameter P_2, and so on, and the n th level of the tree root node each has V_n branches, and the $m+1$ th level is a leaf node.

2) All the values $(v_1, v_2 \cdots v_m)$ taken on the path traversed from the tree root node to the leaf nodes are the solution T of the solution space tree. Where $v_1 \in V_1$, $v_2 \in V_2 \cdots v_m \in V_m$, i.e., v_i is some fetch of the SUT input parameter P_i.

3) For the above SUT with m parameter, set up m vectors with V_i values in $vec[i]$, representing the state of the input parameter P_i after all the values of the input parameter have been arranged according to a certain rule, e.g. $vec[1]=\{a_1,a_2,\cdots a_j\}$, $vec[2]=\{b_1,b_2,\cdots b_j\}\cdots$ and so on.

4) The number of overlaps k of the solution space tree is the number of values that overlap on the paths of the two test cases $T_1=\{v_1,v_2\cdots v_m\}$ and $T_2=\{w_1,w_2\cdots w_m\}$.

It can be seen that the set of all solutions of the solution space tree is the set of test cases that completely combines all the input parameters of the SUT, and each solution corresponds to one piece of test data of the SUT. A test case generation method based on the solution space tree is a method to find a subset of paths in the set of test cases with the smallest size.

II. C. 2) PSST algorithm

Based on the solution space tree model, a solution space tree-based algorithm for generating combinatorial test cases can be realized, and the basic idea is to use the backtracking algorithm to traverse the solution space tree, to find out the set of solutions where the number of overlaps of any test data is not more than 1, and to expand it so that it meets the criteria of combinatorial coverage. Figure 4 shows the flow of the PSST algorithm.

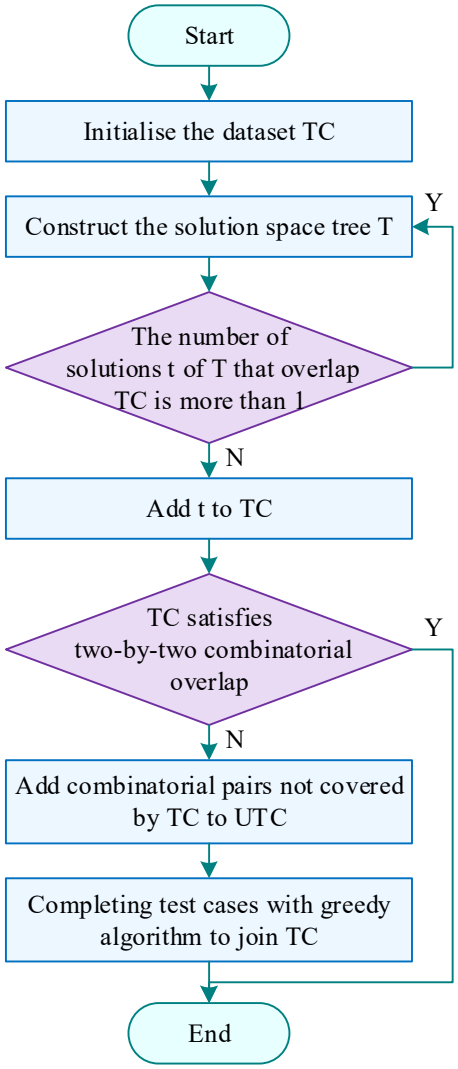

Figure 4: PSST algorithm process

The specific process of the PSST algorithm is as follows:

1) Initialize the test data set TC with $TC\in\varnothing$.

2) Construct the solution space tree T, traverse it depth-first using backtracking, and find the test case t that has no more than 1 overlap with each other with any of the test data in the test set TC and add it to TC. Repeat this step until no solution can be found in the solution space tree that satisfies the condition.

3) Construct a two-by-two ensemble ATC based on all the test data, check whether TC covers all the combinations in ATC, if yes then the algorithm ends, otherwise add the combinations in ATC that are not covered to the set UTC.

4) Generate test cases for the combinations in UTC one by one using the greedy complement algorithm, then add the generated test cases to TC and delete the combinations in UTC that can be covered by this test case, and repeat this step until UTC is empty.

Further, step 4 of the PSST algorithm is further described, at which point the SUT has generated the set of test cases TC and needs to be supplemented by selecting new test cases from the set UTC using a strategy that is a greedy algorithm. The idea is to generate test cases line by line, and the newly selected test cases must cover as many uncovered combinatorial pairs in TC as possible until all combinatorial pairs are covered. The execution process of the greedy algorithm is: after the backtracking algorithm has traversed the solution space tree T, it is necessary to determine whether UTC is empty, if it is empty, it means that all the combinations of any two parameters in the SUT have been traversed by the backtracking method to obtain the set of test cases are completely covered, the test case generation algorithm ends. If it is not empty, then we need to use the greedy algorithm to supplement the set of use cases, select the parameter with the highest number of occurrences in UTC to construct a new test case, to ensure that the new use case can maximize the coverage of the combination of pairs of UTC, and the covered combination of pairs will be deleted, until UTC is empty.

III. Interface testing technology performance testing and effect analysis

III. A. Comparison of the effectiveness of different test case generation methods

III. A. 1) 3.1.1 Comparison and analysis of convergence speed results

Comparison experiments are set up to compare the effectiveness of different test case generation methods. Automatic generation method based on API documentation, generation method based on test data and parameterization, generation method based on packet capturing tool, generation method based on automated testing tool are selected as comparison methods with the PSST algorithm based on solution space tree in this paper. In this, the test rounds are selected up to 50 rounds and the number of test cases is up to 50. Figure 5 gives the variation of the convergence process of the five methods. Comparing the reward values of the five test case generation methods, the final reward value of this paper's solution space tree-based PSST algorithm is 9.524, which is higher than that of the comparison methods, which are 4.856, 6.171, 7.302, and 8.413. During the convergence process of 50 rounds, although this paper's method is not the fastest in terms of convergence speed, it still obtains the highest reward value in the end.

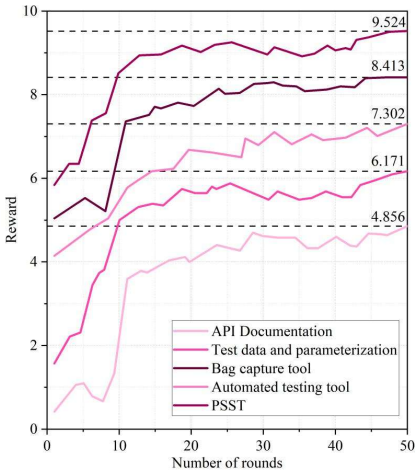

Figure 5: Comparison of convergence results

III. A. 2) 3.1.2 Comparison and analysis of state coverage results

State coverage refers to the number of states that can be covered by the generated test paths. In order to facilitate the comparison, this paper counts the number of test paths that need to be generated by each method under 100% state coverage, and by conversion, we can get the comparison results of different methods with high and low state

coverage. Figure 6 shows the comparison results. In the case of the same state coverage, the PSST algorithm based on the solution space tree in this paper covers all API interfaces with the least number of test paths and generates only 22 test cases. When the comparison method reaches 100% state coverage, 42 test cases are generated by the automatic generation method based on API documents, 43 test cases are generated by the generation method based on test data and parameterization, 37 test cases are generated by the generation method based on packet grabbing tool, and 43 test cases are generated by the generation method based on automated testing tool, which are all more than this paper's algorithm. It shows that the quality of test case generation of this paper's algorithm is better.

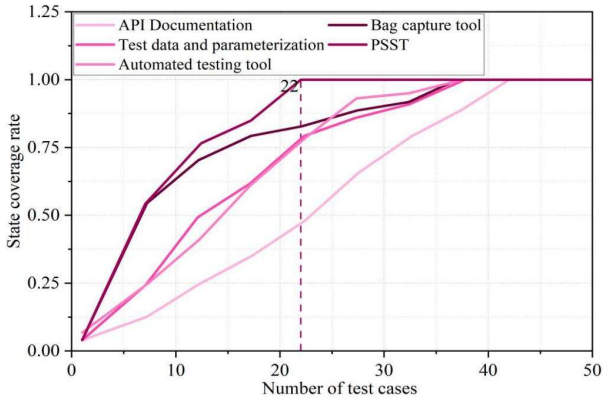

Figure 6: State coverage comparison

III. A. 3) 3.1.3 Comparison and analysis of coverage of key modules

Focused module coverage is the percentage of generated test cases occupied by the focused module. Its basic formula is:

$$mc(k) = \frac{m_k}{k} \tag{5}$$

where $mc(k)$ denotes the coverage of key modules in the case of generating k test case paths, m_k denotes the number of use cases that cover the key modules in the k test case paths generated, and k stands for the number of k use cases generated. The larger value of $mc(k)$ indicates that the test cases cover the key modules more times, i.e., the more important modules can be accessed more than once.

Table 1 summarizes the comparison of the focus module coverage of the five methods. The table mainly lists the focused module coverage of the five methods with the number of generated test cases of 5, 10, 20, 30, 40, and 50 respectively. The PSST algorithm based on the solution space tree in this paper has a higher coverage of key modules in the same number of test cases, which is 0.45, 0.59, 0.78, 0.87 and 1 respectively, and the key modules have a higher chance to be accessed, which can better simulate the user's behavior in the real scenarios, ensure the reliability of interface testing, and improve the efficiency and performance of interface testing. In the case of generating 50 test cases, it even achieves 100% coverage of the key modules, which is higher than the comparison method.

Table 1: Comparison of coverage rates of key modules

Number of test cases	API Documentation	Test data and parameterization	Bag capture tool	Automated testing tool	PSST
5	0.28	0.33	0.37	0.36	0.45
10	0.35	0.39	0.48	0.49	0.59
20	0.42	0.46	0.56	0.52	0.78
30	0.55	0.51	0.67	0.69	0.87
40	0.67	0.69	0.75	0.78	0.95
50	0.71	0.80	0.83	0.86	1

III. B. Analysis of resource utilization in the testing process

III. B. 1) CPU and memory usage

Resource utilization is related to the application cost of the testing process. Reading the test cases generated during the testing process and plotting the corresponding line graphs make it easy to observe the changes in resource utilization during the overall testing process and provide corresponding data for the test report. Figure 7 shows the CPU and memory usage during the running of the test program. During the 150-second interface test, the CPU occupancy fluctuates between 4.12% and 32%, and the memory occupancy continues to rise from 25.44% to 34.62%. On the whole, the resource occupancy of CPU and memory is not high (not more than 45%), which indicates that the algorithm in this paper is characterized by its lightweight.

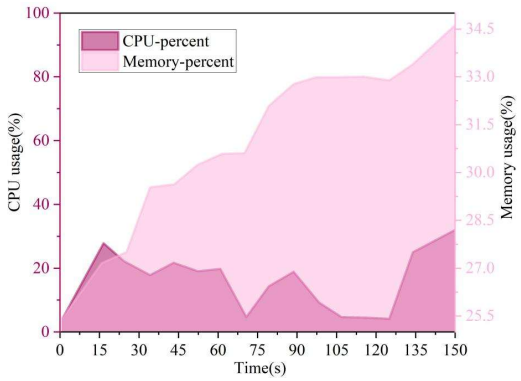

Figure 7: CPU and memory usage

III. B. 2) Thread count and IO read/write variations

Continue to analyze the changes in the number of threads and IO reads and writes during the test program run. Figure 8 shows the specific change process. From the change in the number of threads during the test program run, it can be seen that from the beginning of the unit test program run to open the highest threads of 44.52, with the scheduling of the return to a stable state of work after the drop to about 35.46. From the test program running in the process of IO read and write situation, it can be seen from the unit test program running after the start of the read and write situation grows very quickly, IO read situation from the first second of 2290.72 quickly increased to the 150th second of 57,743.21, IO write situation from the first second of 5565.61 quickly increased to the 150th second of 50,599.55. Especially in the last 25 seconds of the fastest growth rate. .

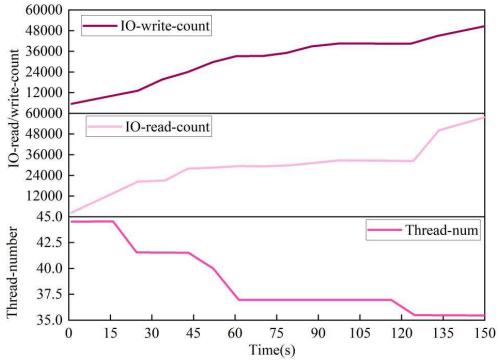

Figure 8: Number of threads and the changes in IO read and write

III. C. Pressurization test result analysis

In order to determine the quality of the interface after the use case generation test, to ensure that the Restful API interface can meet the requirements of a large number of users to access concurrently, this section uses the JMeter testing tool to test the execution of the plan function of the Restful API interface to set a different number of threads to carry out the pressurization test, Table 2 is a summary of the results of the performance test.

The data of the stress test results of different thread counts are recorded, and in the process of simulating concurrent access from 150 to 1050 users, it is clear from the data that the throughput and response time of the Restful API interface are elevated with the increase in concurrency.

Although the system throughput is gradually increasing, but with the increase in the number of threads, the incremental throughput is gradually becoming smaller, when the concurrency of 750 or so, the throughput is basically maintained at about 84 per second, indicating that the platform can meet the needs of 750 simultaneous access.

From the exception column data, we can learn that when the number of threads is less than or equal to 750, the exception rate of the Restful API interface test is 0.00%. When the platform is in the high concurrency (more than 750 simultaneous visits), the Restful API interface test exception rate is 0.17% and 0.29% respectively.

From the response time data, it can be seen that the response time increases with the number of threads, but when the number of threads is less than or equal to 750, the maximum response time is not more than 1.5 s. This indicates that the platform can meet the response time of executing the test plan in the case of high concurrency (750 simultaneous accesses), which is less than 1.5s.

According to the analysis results, after using the PSST algorithm based on the number of solution spaces to generate test cases with 100% coverage and test the Restful API interfaces, the performance level of the Restful API interfaces can be significantly improved, and ultimately meet the requirements of concurrent access for about 750 users.

Table 2: Performance test results

Number of threads	Throughput	Average response time (s)	Minimum response time (s)	Maximum response time (s)	Anomaly(%)
150	43.7/sec	0.02	0.01	0.03	0.00%
300	59.8/sec	0.06	0.01	0.11	0.00%
450	67.5/sec	0.12	0.04	0.20	0.00%
600	84.3/sec	0.46	0.10	0.82	0.00%
750	84.7/sec	0.72	0.19	1.25	0.00%
900	70.2/sec	1.43	0.08	2.78	0.17%
1050	67.3/sec	0.13	0.06	0.19	0.29%

IV. Conclusion

In this paper, we propose a PSST algorithm based on solution space tree to optimize the level of test case generation and improve the efficiency of Restful API interface testing. The algorithm obtains a reward value of 9.524 in 50 rounds of iterations, which is higher than the 4.856, 6.171, 7.302, and 8.413 of the compared methods. 100% state coverage is achieved by generating 22 test cases. 100% key module coverage is achieved by generating 50 test cases. The CPU usage of the testing process is in the range of 4.12%-32% and the memory usage is in the range of 25.44%-34.62%, which is a low resource utilization. When the number of threads is below 750, the tested Restful API interface can meet the demand of simultaneous access. In the future, we can try to consider the abnormal situation in high concurrency scenarios in the test case generation stage, and combine the parameter optimization and other strategies to continuously improve the algorithmic generation of use cases.

References

[1] Mohammed, N. M., Niazi, M., Alshayeb, M., & Mahmood, S. (2017). Exploring software security approaches in software development lifecycle: A systematic mapping study. Computer Standards & Interfaces, 50, 107-115.

[2] Assal, H., & Chiasson, S. (2018). Security in the software development lifecycle. In Fourteenth symposium on usable privacy and security (SOUPS 2018) (pp. 281-296).

[3] Schmidt, M. C., Raman, C. A., Wu, Y., Yaqoub, M. M., Hao, Y., Mahon, R. N., ... & Sun, B. (2021). Application programming interface guided QA plan generation and analysis automation. Journal of applied clinical medical physics, 22(6), 26-34.

[4] Akhilesh, N. S., & Hallikar, R. S. (2022). End to End Representational State Transfer Application Programming Interface Development. International Journal for Research Trends and Innovation, 2456-3315.

[5] Zhong, H., & Mei, H. (2017). An empirical study on API usages. IEEE Transactions on Software Engineering, 45(4), 319-334.

[6] Segura, S., Towey, D., Zhou, Z. Q., & Chen, T. Y. (2018). Metamorphic testing: Testing the untestable. IEEE Software, 37(3), 46-53.

[7] Wulf, J., & Blohm, I. (2020). Fostering value creation with digital platforms: A unified theory of the application programming interface design. Journal of Management Information Systems, 37(1), 251-281.

[8] Jonnada, S., & Joy, J. K. (2019, May). Measure your API complexity and reliability. In 2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA) (pp. 104-109). IEEE.

[9] Srinivas, N., Mandaloju, N., & Nadimpalli, S. V. (2020). Cross-platform application testing: AI-driven automation strategies. Artificial Intelligence and Machine Learning Review, 1(1), 8-17.

[10] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2018). Automated testing of android apps: A systematic literature review. IEEE Transactions on Reliability, 68(1), 45-66.

[11] Kore, P. P., Lohar, M. J., Surve, M. T., & Jadhav, S. (2022). API Testing Using Postman Tool. International Journal for Research in Applied Science and Engineering Technology, 10(12), 841-43.
[12] Sharma, A., & Revathi, M. (2018, November). Automated API testing. In 2018 3rd International Conference on Inventive Computation Technologies (ICICT) (pp. 788-791). IEEE.
[13] Chittineni, S. (2022). Automated API Performance Testing and Anomaly Detection Using Machine Learning in RESTful Architectures. American Journal of AI & Innovation, 4(4).
[14] Arcuri, A. (2020). Automated black-and white-box testing of restful apis with evomaster. IEEE Software, 38(3), 72-78.
[15] Corradini, D., Zampieri, A., Pasqua, M., Viglianisi, E., Dallago, M., & Ceccato, M. (2022). Automated black-box testing of nominal and error scenarios in RESTful APIs. Software Testing, Verification and Reliability, 32(5), e1808.
[16] Ehsan, A., Abuhaliqa, M. A. M., Catal, C., & Mishra, D. (2022). RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 12(9), 4369.