Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Water Loop Software Testing: A Novel Approach for Test Case Generation

Hemant Kumar[1], Vipin Saxena[2]
[1,2]*Department of Computer Science, Babasaheb Bhimrao Ambedkar University, Lucknow, India*

ARTICLE INFO

Received: 29 Dec 2024

Revised: 12 Feb 2025

Accepted: 27 Feb 2025

ABSTRACT

The aim of the present study is to investigate a novel testing approach named Water Loop Software Testing (WLST), which may be used for the automatic generation of test cases, ultimately fostering the development of high-quality software products. An empirical study is meticulously performed, leveraging the concept of a token bucket algorithm for the precise generation of test cases designed to rigorously test software modules. The performance of the proposed technique is then rigorously compared with the established genetic approach of software testing, utilizing a comprehensive case study to ensure a fair and robust evaluation. The computed results, showcasing the efficacy of WLST, are meticulously depicted in the form of detailed tables. It is observed that the presented testing approach generates a substantial 100 test cases in just 0.2670 seconds, accompanied by an impressively low average memory usage of 24.01MB and an average CPU utilization of a mere 0.68%. This remarkable efficiency and resource optimization highlight the novelty of the software testing approach, suggesting that it may be readily adopted by software industries to significantly outperform traditional methods in the development of superior software products.

Keywords: Software Testing, Test Case Generation, Token Bucket Algorithm, Software Development Life Cycle, Water Loop Software Testing.

BACKGROUND

Software product is defined as a finished product which contains independent and dependent software modules which are finally passed through software testing approach. Independent modules are the modules which have no links with another module and it means that the two software modules have no common test cases while on the other hand dependent modules are those modules which have links with another module and obviously having common software test cases. The test cases are explained as an individual or group of parameters/attributes which are used to get optimize results after executing the software modules. The software testing is completely based upon the selection and prioritization of the test cases. For the finished software products, static software testing is very difficult; hence software industries are using the dynamic execution of the test cases for finalizing the software products having high quality and reliability.

Software testing plays a significant role in the Software Development Life Cycle (SDLC) by ensuring the quality of software products through static and dynamic testing methods. Static software testing involves code review method without execution while dynamic software testing examines software behaviour with specific inputs. The generation of the efficient test cases is particularly important in dynamic testing and it is observed from the literature that automation testing has become increasingly popular due to its advantages over static testing. However, the existing literature lacks a comprehensive analysis of generation of the test cases in the optimized time, leading to a research gap. Understanding the execution time involved is crucial for developing effective testing strategies. Therefore, the present study is to address the said research gap by proposing a novel testing approach that will focus on generation of automated test cases in the optimized time. Reduced the time for generation of the test cases has a number of benefits for the development of effective software products. By streamlining this process, software businesses may increase production, reduce expenses, enhance quality, and advertise the products more swiftly.

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

The proposed approach utilizes a token bucket algorithm to automate test case generation and reduce test case generation time. The number of test cases generated per unit of time can be restricted by using token bucket which is a rate-limiting approach. This can help to reduce the amount of time needed for test case generation by preventing the generation of too many test cases at once. The effectiveness and value of this strategy in improving the testing process is evaluated by comparing it with other strategies, such as genetic algorithm. The primary objective of the present study is to generate test cases, speedup the process of doing so, and improve overall effectiveness. Additionally, the model's performance will be analysed by calculating memory usage and CPU utilization. This study aims to fill the current research gap by focusing on the time required for test case generation and conducting a thorough evaluation of the performance of the proposed method. In this approach, the study considerably improves software development, notably in terms of efficiency. The information gathered from this study provides practical suggestions and guidance for improving testing procedures. These insights might lead to enhancement in a number of software development procedures, which would ultimately result in software of greater quality. Let us describe some of the important recent references available in the literature and related to the present work.

In the year 2020, Lakshminarayana and Suresh Kumar [1] proposed an automatic test case generation and optimization technique that utilizes a hybrid cuckoo search and bee colony algorithm. The suggested approach first employs cuckoo search to build initial test cases, which are then improved through bee colony optimization. To assess the effectiveness of the method, several Java programs were tested, and the results demonstrate that it outperformed traditional methods in terms of coverage, efficacy, and efficiency. In another study, Paiva et al. [2] presented a method for generating test cases by modifying user execution traces. The technique modifies user scenario execution and traces to generate new test cases. The proposed approach was evaluated through four distinct case studies, and the results indicate that it generates test cases that are more diverse and effective rather than produced by conventional methods. Another strategy for automated software test case generation, based on an ant colony optimization algorithm, was introduced by Sankar and Chandra [3]. The method employs ant colony optimization to choose and rank test cases based on significance and applicability. The method's performance was evaluated using various benchmark programs, and the results indicate that it outperforms traditional test case generation methods in terms of coverage, efficacy, and efficiency. Further in 2021, Mohd-Shafie et al. [4] conducted an extensive literature review on the generation and prioritization of model-based test cases and discovered several approaches, such as constraint-based testing, model-based testing, and combinatorial testing. Model-based testing was found to be the most commonly used method for test case development, and it was effective in reducing the number of test cases required while maintaining high coverage. Zakeriyan et al. [5] proposed an autonomous test case creation method for industrial software systems based on functional specifications. The approach involved separating testable scenarios from functional specifications using natural language processing and ontology-based approaches. The findings showed that the suggested strategy produced test cases with high coverage. Banerjee et al. [6] also presented an ontology-based approach for automated test case generation. This strategy involved creating a domain-specific ontology to describe the software system being tested and using it to automatic generation of test cases. The research indicated that this method could produce test cases with high coverage while requiring less time and effort than manual test case generation. Sahoo et al. [7] proposed a test case generation approach based on genetic algorithms for UML diagrams. The approach involved for converting UML diagrams to a graph representation to evolve the test cases. The study concluded that the suggested method was effective at generating the test cases with excellent coverage and could be combined with other testing techniques. Wang and Zhao [8] suggested an automated test case generation technique based on an enhanced whale optimization algorithm. The approach encoded test cases as binary sequences and used the enhanced algorithm to search for the best solutions. The study found that the proposed technique successfully produced test cases with high coverage and fewer test cases than required.

In the year 2022, Ma et al. [9] introduced a scalable approach to search for routes that can generate test cases automatically. The proposed method combines a path exploration algorithm and a dynamic slicing technique to produce test cases that cover both critical and non-critical paths of a software system. The path exploration algorithm effectively searches for viable paths, and the results show that it can efficiently generate test cases that achieve high coverage and reveal more flaws as compared to existing methods. Furthermore, the dynamic slicing method reduces the program's size. Further, Pradhan et al. [10] presented an approach for creating test cases for state-chart diagrams utilizing a genetic algorithm. The suggested method creates test cases that incorporate every change in the state chart

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

diagram. The writers evaluated the proposed approach using several case studies and compared it to other state-of-the-art methods. The results show that the approach outperforms existing methods in terms of the coverage achieved and the number of test cases generated. Additionally, the suggested method is effective in discovering software system flaws. In 2023, Rajagopal et al. [11] proposed an automated path-focused test case generation technique for structural program testing using an Adaptive Genetic Algorithm (AGA). The first step involves for constructing a route graph from the source code, followed by generating of the test cases for the graph using AGA. The study showed that the AGA-based test cases achieved better coverage than other traditional test case generation methods. Furthermore, the scope of the test suite was expanded through dynamically parameterizing the generated test cases. Lukasczyk et al. [12] conducted an empirical study to evaluate the effectiveness of Python's automated unit test generation. The research involved ten open-source Python projects and several state-of-the-art test generation tools such as Pynguin, EvoSuite, and TSTL for implementation and comparison. The results demonstrated that automated unit test generation tools produced high-quality tests that were both time and resource efficient and offered adequate coverage. Kumar et al. [13] introduced the Harmony Radial Testing (HRT) approach, combining Harmony Search and Radial Basis Function Neural Network (RBF-NN) to optimize test case generation, improving coverage and fault detection rates compared to existing methods. Based on above and thorough review of research on software testing, it is found that WLST is never studied by the researcher, therefore, the present work is an attempt to investigate a new software testing strategy as illustrated in the Figure 1.

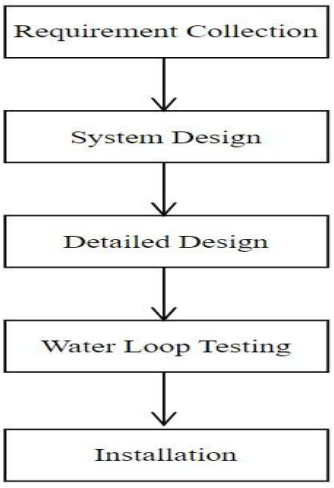

Fig. 1. Water Loop Software Testing during Software Development Phases

METHODOLOGY

A methodology is proposed for automatic generation of the test cases that involves the use of the token bucket algorithm [14] in the water loop testing technique, which is derived from the continuous movement of water in the water cycle. Let us first define the concept of token bucket algorithm. It is a very popular algorithm which is generally used in the field of computer networking for controlling the data flow in the form of traffic between sender and receiver which are well connected across the high speed hybrid network. The data is passed in the form of tokens and if it is valid then forwarded to destination place otherwise discarded the data. It may enhance the efficiency of the computer network. In the present work, tokens are considered as a test case. The definition of the water loop software testing is derived below:

Definition: The water loop testing process is defined as an unceasing movement of test cases on, external and internal components of software modules.

The test case is defined as input parameters used to get the expected output from the software module. For the sake of clarification, a software module is considered as a bucket which may consist of many sub-modules. The software product (M) is defined as a collection of the software modules [M_1,M_2,...,M_j.....M_n]. In the software product, some software modules may be independent while others may be dependent software modules. The mathematical representation is given below:

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

$$M \equiv [M_1, M_2, \ldots, M_j \ldots M_n] \quad (1)$$

Where, M_1, M_2,..., M_j are considered as independent modules and M_{j+1},...., M_n are dependent modules. The software module executes on various test cases which are categorized as valid or invalid test cases. The valid test case is defined as input parameters which are producing the correct results and while invalid test cases are not producing the correct results. Over the software module, the test cases are further categorized according to the followings:

Internal Test Case (ITC): It is explained as a test case on which the module is currently executing which may also be defined inside the software module.

$$ITC \equiv [T_1, T_2, \ldots \ldots T_j\} \quad (2)$$

Boundary Test Case (BTC): It is explained as test case which is trying to get a software module for execution purpose. When such a test case sets then it may be converted into an internal test case.

$$BTC \equiv \{T_{j+1}, T_{j+2}, \ldots \ldots T_k\} \quad (3)$$

External Test Case (ETC): It is explained as test case outside the software module that may be converted into a boundary test case and when these are executed then converted into the internal test case.

$$ETC \equiv \{T_{k+1}, T_{k+2}, \ldots \ldots T_n]$$

Hence, the total test cases are $T \equiv ITC \cup BTC \cup ETC$. This concept is depicted in the following Figure 2.

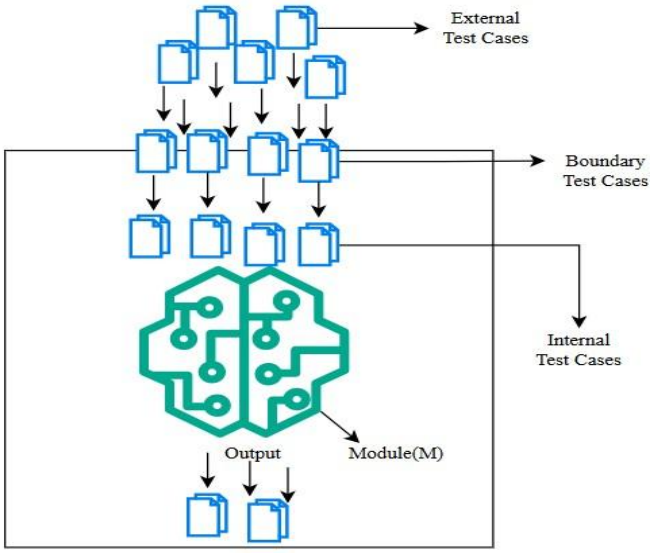

Fig. 2. Representation of the Various Test Cases

The proposed methodology may be applied for the four standard phases of software development like collection of feasibility analysis, requirements collection, system design, detailed design and coding via WLST to generate efficient software products over the valid test cases, as shown in the following Figure 3.

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

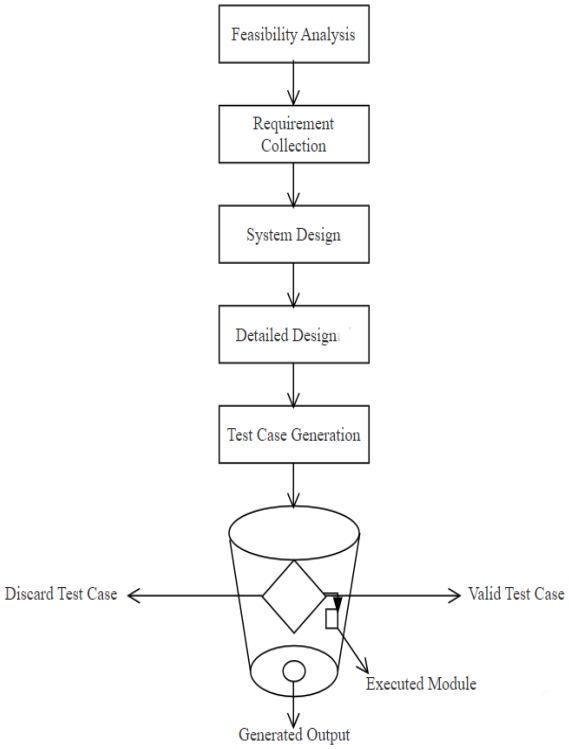

Fig. 3. Generation of Valid Test Cases

Traditional manual testing methods used in software development may be time-consuming and may not result in generating effective test cases. In the presented approach, the main aim is to reduce the execution time considered for generation of test cases which obviously improve coverage criteria. The novel token bucket algorithm is generating automated test cases. It regulates the flow of data by defining a bucket with a fixed capacity that fills up with tokens at a set rate. The main aim of proposed approach is to reduce the time taken to generate test cases and improve the efficiency of the testing process. By using the token bucket algorithm and water loop testing approach, one may generate more effective test cases. The concept for generation of valid test case and overflow condition are also presented in the following figure 4.

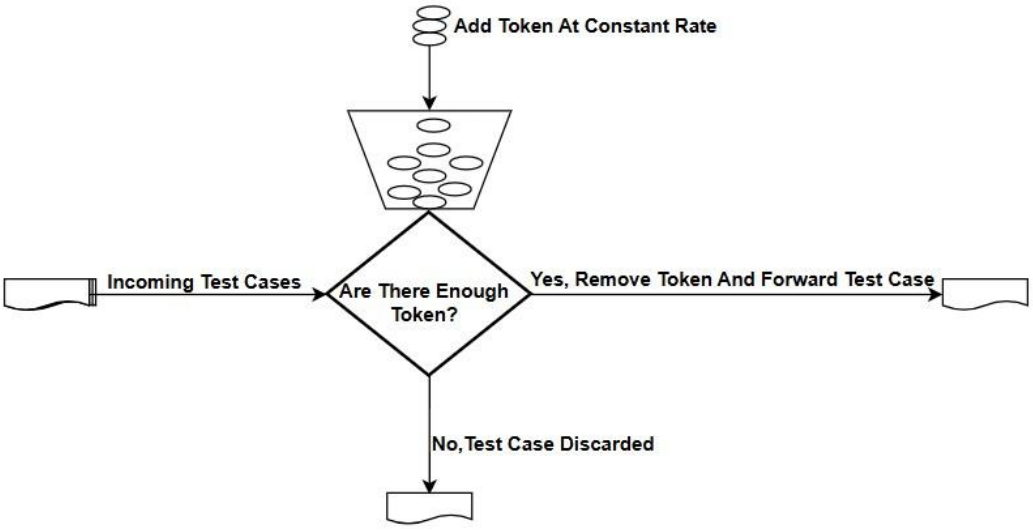

Fig. 4. Overflow Representation of Test Cases

On the basis of above concept, the WLST algorithm is given below:

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

WLST()

S → size of software finished product

N → number of modules

r → generation rate of test cases

t_s →start time of test case entering for execution

t_e → end time of test case after used by the software module

test time → t = t_e-t_s

*A=min(A+(t-t_l)*r,S) //test case replenishment equation*

condition:A>=1, software module is tested over test case

otherwise, software module is generating wrong results, so test case is discarded

RESULTS

Let us consider a case study which describes a module used in the computation of the natural frequencies for transverse vibration of the skew plates used in the mechanical chips, naval structure, air craft designs, etc. The valid test cases are generated through WLST approach over the involvement of the integrals used for computation of the frequencies. A skew plate is represented in the following figure 5 which is converted into square plate using the transformation approach of variables converted from (x,y) to (ξ, η).

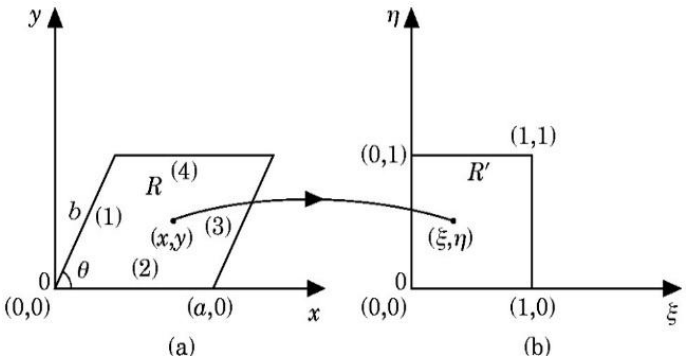

Fig. 5. A Representation of Skew Plate [15]

The first three natural frequencies λ of skew plate may be obtained using eigen-value problem as represented below:

$$\sum_{J=1}^{N}(a_{ij} - \lambda^2 b_{ij})c_j = 0, \quad i = 1,2, \dots\dots\dots\dots, N \tag{4}$$

The above equation is called as eigen-value problem and has lengthy computations of the various integrals; therefore, WLST concept is applied for effective execution of the integrals through automatic generation of the valid test case. The involvement of integrals may be seen from the following matrices given in the equations 5 and 6 which are used to provide the various values of λ:

$$aij = \frac{1}{sin^4\theta}\iint_{R'} f3\left[\Phi_i^{\xi\xi}\phi_j^{\xi\xi} - 2\mu\cos(\theta)\left(\phi_i^{\xi\eta}\phi_j^{\xi\xi} + \phi_i^{\xi\xi}\phi_j^{\xi\eta}\right) + \mu^2\left(v\,sin^2(\theta) + cos^2(\theta)\right)\left(\Phi_i^{\eta\eta}\phi_j^{\xi\xi} + \phi_i^{\xi\xi}\phi_j^{\eta\eta}\right) + 2\mu^2(1 + cos^2\theta) - vsin^2(\theta)\phi_i^{\xi\eta}\phi_j^{\xi\xi} - 2\mu^3\cos(\theta)\left(\phi_i^{\eta\eta}\phi_j^{\xi\eta} + \phi_i^{\xi\eta}\phi_j^{\eta\eta}\right) + \mu^4\phi_i^{\eta\eta}\phi_j^{\eta\eta}\right]d\xi d\eta \tag{5}$$

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

$$b_{ij} = \iint_{R'} f\phi_i\phi_j \, d\xi \, d\eta \qquad (6)$$

where,

$$\lambda^2 = \frac{(12)\left(1-\frac{V^2\rho a^2 m^2}{Eh^2}\right)}{\frac{a}{b}}, \quad f(\xi,\eta) = (1+a\xi)(1+\beta\eta),$$

7)

and

$$\phi_i(\xi,\eta) = \xi^p\eta^q(1-\xi)^r(1-\eta)^s(1,\xi,\eta,\xi^2,\xi\eta,\eta^2,\ldots\ldots) \qquad (8)$$

The above computation of natural frequencies involves the following integration:

$$\iint \xi^{LP}\eta^{LQ}(1-\xi)^{LR}(1-\eta)^{LS}d\xi\, d\eta = \frac{LP!LQ!LR!LS!}{(LP+LR+1)!(LQ+LS+1)!} \qquad (9)$$

WSLT is used to compute valid test cases from the equation (9) for computation of first three natural frequencies for skew plate.

RESULTS AND DISCUSSION

In this study, the tokens are generated through token bucket algorithm in which the **init** method initializes the following class and instance variables. The **generate ()** method calculates the current time and the time difference since the last update. Based on the difference and the **time_unit**, it adds tokens to the bucket and updates the **last_update** time. If the token value is less than **1**, it returns false, indicating that there are no tokens available, otherwise, it decreases the token count by **1** and increases the counter. If the counter is equal to bucket capacity, it resets the counter to **0** and returns False, otherwise, it returns True, indicating that a token is available. The concept is given below:

```
class TokenBucket:
  def __init__(self, token, capacity, time_unit):
    self.token = token
    self.capacity = capacity
    self.time_unit = time_unit
    self.counter = 0
    self.last_update = time.time()

  def generate(self):
    current_time = time.time()
    time_since_last_update = current_time - self.last_update
    tokens_to_add = time_since_last_update / self.time_unit
    self.token = min(self.token + tokens_to_add, self.capacity)
    self.last_update = current_time

    if self.token < 1:
      return False
    else:
```

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
        self.token -= 1
        if self.counter == self.capacity:
            self.counter = 0
            return False
        self.counter += 1
        return True
```

The generation of test cases is described through following segment of code nearer to Python programming language:

```
def generate_testcases():
    tb = TokenBucket(100, 100, 1)
    test_cases = []
    for i in range(1,128):
      if tb.generate():
        AL = 1.0
        NO = 2
        LP = random.choice([0, 1, 2, -1])
        LQ = random.choice([0, 1, 2, -1])
        LR = random.choice([0, 1, 2, -1])
        LS = random.choice([0, 1, 2, -1])
        test_cases.append((AL, LP, LQ, LR, LS, NO, HT(AL, LP, LQ, LR, LS, NO)))
    return test_cases
```

The above function performs the following steps:

Step1: *Instantiates a **TokenBucket** object with an initial token count of 10**0**, a bucket capacity of 10**0**, and a time unit of **1**;*

Step2: *Creates an empty list called **test_cases** to store the generated test cases;*

Step3: *Loops through a range of values from 1 to 128;*

Step4: *On each iteration of the loop, it calls the **generate ()** method of the TokenBucket object to determine if a token is available;*

Step5: *If a token is available, it uses the **random.choice** function to randomly select values for the variables **AL, LP, LQ, LR**, and **LS**;*

Step6: *The value of **NO** is set to **2**, and the function **HT** is called with these values as arguments;*

Step7: *The result of the **HT** function call is appended to the **test_cases** list along with the arguments passed to the function;*

Step8: *After all iterations of the loop, the **test_cases** list is returned.*

The above function generates 100 test cases, which is the capacity of a bucket considered as finite capacity, when bucket is full of test cases, and then one can apply the WLST to check whether the test case is valid or invalid. The procedure for generation of valid test cases is given below:

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

```
invalid_test_cases = []
valid_test_cases=[]
@pytest.mark.parametrize("AL, LP, LQ, LR, LS, NO, expected_output", generate_testcases())
def test_HT(AL, LP, LQ, LR, LS, NO, expected_output):
  if LP < 0 or LQ < 0 or LR < 0 or LS < 0:
  invalid_test_cases.append((AL, LP, LQ, LR, LS, NO, expected_output))
     assert expected_output == None
   else:
  valid_test_cases.append((AL, LP, LQ, LR, LS, NO, expected_output))
    assert expected_output != None
```

The above code is using the pytest library to perform a set of tests on the function **HT**. The tests are performed using the **@pytest.mark.parametrize** decorator, which allows a set of test cases to be run with the same function.

1. The **valid_test_cases** and **invalid_test_cases** lists are created to store valid and invalid test cases, respectively;
2. The **@pytest.mark.parametrize** decorator is applied to the **test_HT** function and is used to run a set of test cases using the function. The parameters for the test cases are generated using the **generate_testcases()** function;
3. In the **test_HT** function, the first if statement checks if any of the values **LP, LQ, LR,** or **LS** are less than **0**. If any of these values are negative, the test case is considered invalid and is added to the invalid_test_cases list. The assert statement verifies that the expected output is equal to **None**;
4. If all of the values **LP, LQ, LR,** and **LS** are positive, the test case is considered valid and is added to the valid_test_cases list. The assert statement verifies that the expected output is not equal to **None.**

The process of generating test cases has several stages, which are depicted in the following figure 6 along with their interactions.

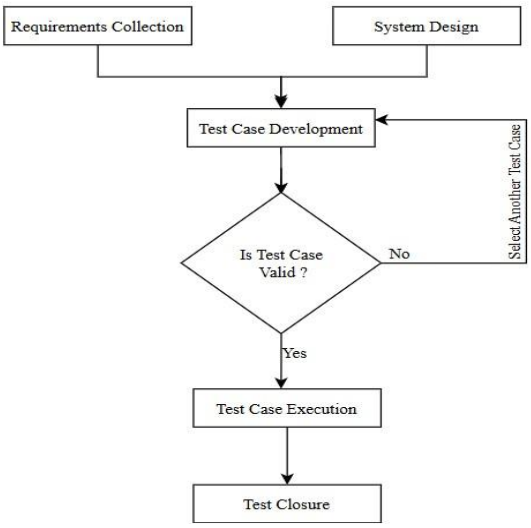

Fig. 6. Test Case Execution in Water Loop Testing

We conducted a comparison between our proposed water loop testing technique and the genetic algorithm, which is a widely used test case generation technique in software testing. The genetic algorithm generated a total of 100 test

Journal of Information Systems Engineering and Management
2025, 10(57s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

cases and took 19.6601 seconds to complete, while our approach generated the same number of test cases in just 0.2670 seconds. This result clearly demonstrates the effectiveness of our approach in reducing the execution time required to generate test cases. The elapsed time required to generate test cases is determined by calculating the time that has passed between the start and end of a particular event, process, or task. This time is typically measured in units such as seconds, minutes, hours, or even days and is calculated as the difference between the end time and the start time of the event, process, or task. Based on Table 1, which shows the comparison results between the genetic algorithm and water loop testing, our approach clearly outperformed the genetic algorithm in terms of the time taken to generate test cases.

Table 1. Comparison of Genetic Algorithm and WLST

Test Cases	Genetic algorithm	WLST
20	4.2264	0.1512
40	7.3255	0.1536
60	12.9831	0.1560
80	15.4930	0.1682
100	19.6601	0.2670

while Table 2 provides a numerical comparison of memory usage between the existing genetic algorithm and our WLST approach.

Table 2. Memory Usage-WLST versus GA

Iteration	WLST Memory Usage (MB)	GA Memory Usage (MB)
1	23.96	24.31
2	23.97	24.32
3	23.98	24.39
4	23.98	24.41
5	23.99	24.42
6	24.00	24.44
7	24.00	24.44
8	24.00	24.52
9	24.05	24.52
10	24.09	24.52

The results indicate that our WLST model exhibited an average memory usage of 24.01MB with an average CPU utilization of 0.68%. In contrast, the genetic algorithm technique resulted in an average memory usage of 24.436MB with a CPU utilization of 0.77%. The disparities in memory usage can be attributed to the underlying algorithms and methodologies employed by each technique. The WLST technique leverages the token bucket algorithm, which effectively manages resource allocation, leading to optimized memory utilization. On the other hand, the genetic algorithm technique may involve more complex operations, resulting in relatively higher memory consumption. To ensure the observed performance improvements of the WLST algorithm over the Genetic Algorithm (GA) are

statistically significant, we conducted an ANOVA test on execution time and memory usage. The results of the ANOVA test are presented in Table 3.

Table 3. ANOVA Test Results for WLST and GA

Metric	F-statistic	p-value	Significance
Execution Time	17.94	0.00285	Statistically Significant
Memory Usage	244.74	6.37×100^{-12}	Highly Statistically Significant

The ANOVA test results indicate that the differences in both execution time and memory usage between WLST and GA are statistically significant. These results confirm the superiority of the proposed WLST algorithm, which generated test cases faster than GA while utilizing approximately less memory. The WLST algorithm significantly outperforms the GA in both execution time and memory usage. The ANOVA test validates these differences, highlighting WLST's superior efficiency and resource optimization for test case generation.

CONCLUSIONS

WLST a novel software testing technique designed to reduce the time required for generating test cases, is presented in the current study. Reducing test case generation time is beneficial for the development of efficient software products, such as faster test case generation, more testing time, better software quality, and ultimately improved customer satisfaction. Our Water Loop Testing (WLST) approach generates 100 test cases in 0.2670 seconds, compared to the Genetic Algorithm (GA) approach, which takes 19.6601 seconds. WLST utilizes an average memory of 24.01MB and a CPU utilization of 0.68%, while the genetic algorithm technique utilizes 24.436MB of memory and has a CPU utilization of 0.77%. It fills a research gap related to test case generation time and advances software development practices. The Water Loop Testing (WLST) strategy, integrated with the token bucket algorithm, shows promise in generating dynamic test cases, ensuring software quality while reducing development time. Further research can explore Water Loop Testing (WLST) scalability and applicability in different software development scenarios and evaluate the effectiveness of the proposed approach on larger programs and in finding bugs.

REFRENCES

[1] P. Lakshminarayana and T.V. Suresh Kumar, “Automatic generation and optimization of test case using hybrid cuckoo search and bee colony algorithm,” *Journal of Intelligent Systems*, Vol. 30 No. 1, pp.59-72, https://doi.org/10.1515/jisys-2019-0051.

[2] A.C. Paiva, A. Restivo, A. and S. Almeida, “Test case generation based on mutations over user execution traces,” *Software Quality Journal*, Vol. 28, pp.1173-1186 2020, https://doi.org/10.1007/s11219-020-09503-4.

[3] S. Sankar, and V.S.S. Chandra, “An ant colony optimization algorithm based automated generation of software test cases,” In *Advances in Swarm Intelligence: 11th International Conference, ICSI 2020, Belgrade, Serbia, July 14–20, 2020, Proceedings Vol. 11,* pp. 231-239, Springer International Publishing, https://doi.org/10.1007/978-3-030-53956-6_21.

[4] M.L. Mohd-Shafie, W.M.N.W. Kadir, H. Lichter, M. Khatibsyarbini, and M.A. Isa, “Model-based test case generation and prioritization: a systematic literature review,” *Software and Systems Modeling*, pp.1-37, 2021, https://doi.org/10.1007/s10270-021-00924-8.

[5] A. Zakeriyan, R. Khosravi, H. Safari, and E. Khamespanah, “Towards automatic test case generation for industrial software systems based on functional specifications,” In *Fundamentals of Software Engineering: 9th International Conference, FSEN 2021, Virtual Event, May 19–21, 2021, Revised Selected Papers, Vol. 9*, pp. 199-214, 2021, Springer International Publishing. https://doi.org/10.1007/978-3-030-89247-0_14.

[6] S. Banerjee, N.C. Debnath, and A. Sarkar, ”An Ontology-Based Approach to Automated Test Case Generation,” *SN Computer Science*, Vol. 2, pp.1-12, 2021, https://doi.org/10.1007/s42979-020-00420-8.

[7] R.K. Sahoo, M. Derbali, H. Jerbi, D. Van Thang, P.P. Kumar, and S. Sahoo, “Test Case Generation from UML-Diagrams Using Genetic Algorithm” *CMC-COMPUTERS MATERIALS and CONTINUA*, Vol. *67, No.*2, pp. 2321-2336, 2021, https://doi.org/10.32604/cmc.2021.013014.

[8] J. Wang, and W. Zhao, "Automatic Test Case Generation Method Based on Improved Whale Optimization Algorithm," In *2021 5th International Conference on Intelligent Systems, Metaheuristics and Swarm Intelligence*, pp. 7-16, April 2021, https://doi.org/10.1145/3461598.3461600.

[9] E. Ma, X. Fu, and X. Wang, (2022) "Scalable path search for automated test case generation," *Electronics*, Vol.11, No.5, pp. 1-22, 2022, https://doi.org/10.3390/electronics 11050727.

[10] S. Pradhan, M. Ray, and S.K. Swain, "Transition coverage -based test case generation from state chart diagram," *Journal of King Saud University-Computer and Information Sciences*, Vol. *34, No.*3, pp. 993-1002, 2022, https://doi.org/10.1016/j.jksuci. 2019.05.005.

[11] M. Rajagopal, R. Sivasakthivel, K. Loganathan, L.E. and Sarris, "An Automated Path-Focused Test Case Generation with Dynamic Parameterization Using Adaptive Genetic Algorithm (AGA) for Structural Program Testing," *Information*, Vol. *14, No.* 3, pp. 1-18, 2023, https://doi.org/10.3390/info14030166.

[12] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, Vol. *28, No.*2, pp.1-46, 2023, https://doi.org/10.1007/s10664-022-10248-w.

[13] H. Kumar, and V. Saxena, "Effective test cases generation with harmony search and RBF neural network," *The International Arab Journal of Information Technology (IAJIT)*, vol. *21*(5), pp. 786–799, 2024, DOI: https://doi.org/10.34028/iajit/21/5/2.

[14] T.C. Tsai, C.H. Jiang, and C.Y. Wang, "CAC and packet scheduling using token bucket for IEEE 802.16 networks," *J. Communication.*, Vol. 1, No. 2, pp. 30-37, 2006, https://doi.org/10.4304/jcm.1.2.30-37.

[15] B. Singh, and V. Saxena, "Transverse vibration of skew plates with variable thickness," *Journal of Sound and Vibration*, Vol. 206, No.1, pp.1-13, 1997, https://doi.org/10.1006/jsvi.1997.1032.