Nanotechnology Perceptions
ISSN 1660-6795
www.nano-ntp.com

A Novel Algorithm For Generating Hard-To-Kill Higher Order Mutants Using Chemical Reaction Optimization

Subhasish Mohanty[1], Jyotirmaya Mishra[2], Sudhir Kumar Mohapatra[3,*], Melashu Amare[4]

[1,2] *Computer Science and Engineering, GIET University, Gunupur, Odisha.*
[3]*Faculty of Engineering & Technology, Sri Sri University, Bhubaneswar ,Odisha.*
[4] *Departments of Software Engineering, Woldia University, Woldia, Ethiopia.*

In the field of software engineering, ensuring the reliability and robustness of software is paramount, and software testing plays a critical role in this process. Mutation testing, a fault-based technique, evaluates the effectiveness of test suites by introducing artificial defects, known as mutants, into programs. This research presents a novel method for generating higher-order mutants (HOMs) using the Chemical Reaction Optimization (CRO) algorithm, which enhances the rigor of mutation testing by creating harder-to-detect mutants. The CRO algorithm employs four types of collision operators: on-wall ineffective, synthesis, decomposition, and inter-molecular ineffective, to modify mutants and simulate complex faults. Through experimentation with iterations set at 10, 30, and 50, it was found that increasing the number of iterations significantly reduces the number of mutants and increases their detection difficulty. Notably, with 50 iterations, the approach achieved a 93% reduction in mutants and lowered the mutation score to 27.77%, demonstrating the robustness of the generated mutants. The research further introduces the HOMUsingCRO tool, which automates the mutant generation and testing process, generating XML-based reports for effective mutant analysis. The proposed approach outperforms existing techniques in both mutant reduction and mutation score, offering a more comprehensive solution for improving software test suite effectiveness.

Keywords : Real fault, hard to detect mutant, chemical reaction optimization algorithm, mutation testing, higher-order mutant generation, unit testing.

1. Introduction

Software engineering is a discipline dedicated to the systematic design, development, and maintenance of software systems. As these systems grow in complexity, ensuring their reliability becomes a crucial challenge. Software testing is an integral part of this process, serving as a key method for validating the functionality and robustness of software applications. Mutation testing, one of the more advanced techniques, evaluates the quality of a test suite by introducing artificial faults, or mutants, into the program. These mutants, which are slight modifications to the original code, simulate real-world defects that could arise during software development. The effectiveness of the test suite is measured by its ability to "kill"

these mutants—i.e., detect and eliminate them—thereby assessing the robustness of the software in detecting and handling potential faults. [1].

There are two primary types of mutants used in mutation testing: First-Order Mutants (FOMs) and Higher-Order Mutants (HOMs). FOMs involve single, localized changes to the program, while HOMs are the result of combining multiple FOMs [2]. Although FOMs are easier for a test suite to detect, HOMs are more resistant to detection, simulating more complex and subtle faults [3]. This makes HOMs a more rigorous test of a suite's fault-detection capability. However, generating and managing HOMs is more challenging due to the exponential increase in possible combinations and the complexity of the mutants produced [4].

Metaheuristic algorithms, particularly those inspired by natural processes, have shown great promise in optimization problems across various fields, including software testing [5]. Chemical Reaction Optimization (CRO) is a relatively new metaheuristic algorithm that mimics chemical reactions in nature, where molecules interact and change states to reach stable conditions [6]. In the context of mutation testing, the CRO algorithm transforms software mutants through a series of reactions, aiming to generate harder-to-kill HOMs. By randomly applying four different collision types—synthesis, decomposition, on-wall ineffective, and inter-molecular ineffective—the CRO algorithm can either increase or reduce the number of mutants, while also enhancing their resilience to detection by the test suite [7][8].

This study introduces a novel approach for generating higher-order mutants using the CRO algorithm. The proposed method improves the complexity and difficulty of detecting these mutants, which in turn enhances the robustness of the software's test suite. The authors applied the CRO algorithm at different iteration levels 10, 30, and 50 observing the impact on the number of mutants generated and their detection rates. The results indicate that increasing the number of iterations significantly reduces the number of mutants and increases their difficulty of detection, making the test process more rigorous and reliable.

The contributions of this research are threefold:

1. Introduced a new methodology for generating higher-order mutants using CRO.
2. Evaluated the impact of different iteration counts on mutant generation and detection difficulty, and
3. Developed a tool, HOMUsingCRO, which automates the process of generating, executing, and analyzing mutants using CRO. This tool generates XML-based reports, making the mutant analysis process more efficient. The findings show that the proposed approach outperforms existing techniques in both mutant reduction and mutation score, demonstrating the practical applicability of our methodology for improving software testing processes.

The organization of this article is as follows: the next section discusses the related work.

In section 3, a detailed discussion on the proposed methodology is presented. Section 4 contains the implementation of the proposed methodology. The results and discussion are presented in section 5. The article is concluded in section 6.

2. Related Work

The generation of higher-order mutants (HOMs) has been extensively studied as a means to improve the effectiveness of mutation testing. Various techniques have been proposed to generate HOMs that are harder to detect and more representative of real-world faults.

Anas Abuljadyel et al. [9] introduced an agent-based algorithm combined with a genetic algorithm (GA) to create subtle higher-order mutants. Their method initially achieved a 50% mutation score, but after incorporating the GA, the score was reduced to 0.2% among 2000 mutants and 800 test cases. Despite the reduction in mutation scores, the study did not achieve significant reduction in the number of generated HOMs, indicating that their approach, while effective in generating challenging mutants, lacked efficiency in mutant reduction. Elmahdi Omar et al. [4] proposed three novel metaheuristic search strategies—guided local search, restricted enumeration, and restricted local search—to generate higher-order mutants. These strategies outperformed traditional methods like genetic algorithms, local search, and random search in terms of producing more difficult-to-detect HOMs. However, their approach did not focus on reducing the overall number of mutants, limiting its practical applicability in large-scale mutation testing scenarios.

Mike Papadakis et al. [10] conducted a large-scale empirical study exploring the correlation between mutation scores and real fault detection. Their findings revealed that mutation scores strongly correlate with a test suite's ability to detect real faults, underscoring the importance of mutation testing for evaluating test suite quality. However, their study primarily concentrated on first-order mutants, leaving higher-order mutants underexplored in terms of their potential to simulate more complex faults. Nguyen et al. [1] evaluated multi-objective optimization algorithms aimed at generating HOMs. Their approach successfully balanced the trade-off between mutant generation and test case effectiveness. However, the scope of their study was constrained by relatively small subject programs, limiting its ability to assess the scalability of their optimization techniques for larger software systems.

Nishtha Jatana et al. [12] developed an improved Crow Search Algorithm (ICSA) to enhance the automation of test case generation for mutation testing. While this approach demonstrated strong performance in generating optimized test suites, it primarily addressed test data generation rather than focusing on generating challenging HOMs. As a result, it fell short in tackling the complexity of HOM creation. An SSHOM tool was proposed in [13] to perform first-order mutation (FOM) testing on selected pairs of mutants and then combine the resulting FOMs to generate higher-order mutants (HOMs). This tool provided an efficient approach to systematically combining mutations, contributing to the generation of complex HOMs. In [14], the author introduced a scheme for identifying subsuming higher-order mutants (SOMs), which are mutants that subsume others, to help reduce the total number of required mutations

while maintaining the testing effectiveness. This scheme helped streamline the mutation process by focusing on key mutants that cover more fault scenarios.

A comprehensive literature review in [15] enhanced the understanding of HOMs by examining various approaches and challenges associated with generating and killing HOMs. This review provided valuable insights into current methodologies and identified future research directions. In [16], the author explored the application of multi-objective optimization algorithms for solving real-time problems related to HOM testing. This study demonstrated the practical utility of HOMs in complex, real-world scenarios and highlighted the advantages of using optimization algorithms to improve testing efficiency.

The author in [17] addressed two critical challenges: identifying the most suitable SOMs for testing and developing methods to generate hard-to-kill mutants. These hard-to-kill mutants offer greater insight into the effectiveness of test suites, making them highly valuable for rigorous testing. Further studies [18, 19, 20] investigated the use of soft computing techniques, such as genetic algorithms, to effectively kill HOMs. These approaches demonstrated the potential of leveraging advanced algorithms to improve the overall effectiveness of mutation testing. In [21], the author evaluated mutation testing based on FOMs and identified high-quality mutation operators that could be used to generate more effective HOMs. This approach contributed to the refinement of mutation strategies for improved efficiency and accuracy. Other studies have explored the use of mutation testing for various purposes, including software testing [22,23,24,25,26], test case prioritization [27,28,29], and test case reduction [30]. These contributions have expanded the scope of mutation testing, demonstrating its versatility and effectiveness in different software testing contexts.

Table 1: Summary of LR

Author(s)	Year	Methodology/Technique	Problem Addressed	Key Findings	Limitations
Anas Abuljadyel et al.	2018	Agent-based algorithm with genetic algorithm (GA) integration	Creation of subtle higher-order mutants resistant to tests	Mutation score reduced to 0.2% with 2000 mutants, 800 test cases	No significant reduction in generated HOMs
Elmahdi Omar et al.	2014	Guided local search, restricted enumeration, restricted local search	Generating higher-order mutants with minimal cost	These approaches outperformed GA, local, and random search	Did not address reducing mutant population size significantl

					y
Mike Papadakis et al.	2018	Mutation testing correlated with real faults	Evaluating if mutation scores correlate with real fault detection	Showed strong correlation between mutation scores and fault detection	Limited exploration of HOMs
Nguyen et al.	2016	Multi-objective optimization algorithm for HOM testing	Searching for higher-order mutants through optimizatio n	Successfully generated and reduced HOMs with improved test case effectiveness	Large-scale subject programs were not fully tested
Nishtha Jatana et al	2020	Improved Crow Search Algorithm (ICSA)	Test data generation using mutation testing	Efficient in generating optimized test suites automaticall y	Lacks focus on the generation of complex HOMs
Shin Yoo et al	2021	SSHOM tool for performing FOM and generating HOMs	Combining FOMs to generate higher-order mutants	Systematic approach to generating HOMs from FOMs	Limited to tool-based application
Eduardo Figueiredo et al	2021	Scheme for identifying subsuming higher-order mutants (SOMs)	Reduction of mutations by identifying key mutants	Reduced mutations by identifying key SOMs	Focused primarily on subsuming mutants

While prior studies have made significant advancements in generating and optimizing higher-order mutants, they have generally focused on either increasing mutant complexity or reducing mutant numbers, but rarely on addressing both aspects simultaneously. Techniques like genetic algorithms and multi-objective optimization have been effective in creating hard-to-detect mutants but often fall short in reducing the number of mutants. Conversely, methods aimed at mutant reduction frequently do not produce complex or challenging mutants. Existing research has tended to prioritize either detection difficulty or mutant reduction, often

overlooking the need for a balance between the two. Additionally, many studies have been evaluated on small-scale subject programs, limiting their applicability to larger, more complex software environments. This study addresses these gaps by introducing a Chemical Reaction Optimization (CRO) algorithm that simultaneously reduces the number of higher-order mutants and increases their detection difficulty, offering a more comprehensive and rigorous approach to mutation testing.

3. Methodology

Chemical Reaction Optimization (CRO) is a versatile metaheuristic algorithm designed to solve a variety of optimization problems. Unlike algorithms tailored to specific domains, CRO operates independently of problem specifics, allowing for customization based on research requirements [7]. The methodology consists of three main phases: Initialization, Iteration, and Finalization.

3.1 Initialization Phase

During the initialization phase, each mutant, whether first-order or higher-order, is represented as a molecule. This phase involves setting up the algorithm's parameters, which include **genSize**, **buffer (b), alpha**, and **beta**. The parameter **genSize** specifies the stopping condition for the algorithm, with the user determining the number of iterations the algorithm will execute. The **buffer** is a random value between [0, 1] used to distinguish between uni-molecular and inter-molecular collisions. The **alpha** value, also a random number between [0, 1], is used to decide if an inter-molecular collision results in synthesis or an ineffective collision. Similarly, the **beta** value determines whether a uni-molecular collision leads to decomposition or an on-wall ineffective collision.

3.2 Iteration Phase

In the iteration phase, the algorithm executes based on the stopping condition specified by **genSize**. During this phase, collisions can be classified as either uni-molecular or inter-molecular. The type of collision is determined by generating a random **buffer** value. If this value exceeds 0.5, an inter-molecular collision occurs; otherwise, a uni-molecular collision is performed.

For inter-molecular collisions, the process involves generating a random **alpha** value to decide between an ineffective collision and synthesis. In synthesis, two mutants with average fitness values are selected from each module and merged to reduce their number. The ineffective collision utilizes crossover operations, similar to those in genetic algorithms, to enhance the search space.

For uni-molecular collisions, a random **beta** value determines whether the collision results in an on-wall ineffective collision or decomposition. In the decomposition process, a mutant with the highest number of hits is divided into two mutants, increasing the search space. The on-wall ineffective collision involves selecting a mutant with lower fitness and applying

traditional and class-level operators to alter its structure, analogous to mutation in genetic algorithms.

3.3 Finalization Phase

Once the stopping condition is met, the algorithm moves to the finalization phase. Here, the solution is saved in an XML file on the disk. The stopping criterion is defined by the user and can be specified through a Java input dialog. The final phase concludes with the output of the solution, encapsulated in XML format, representing the result of the optimization process.

In summary, the CRO algorithm operates through the initialization of parameters, iterative collision processing, and finalization of the solution [7].

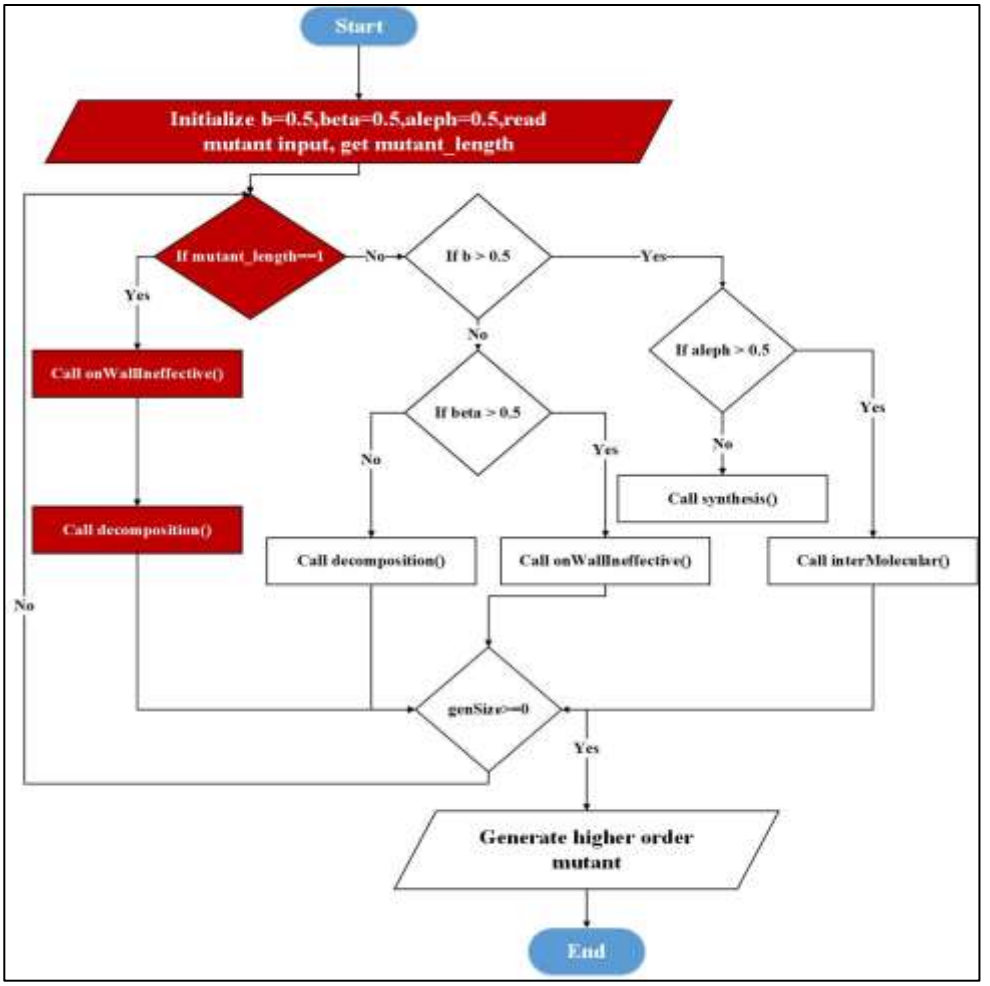

Fig. 1. Proposed CRO algorithm flowchart

The flow of the Chemical Reaction Optimization (CRO) algorithm is outlined as follows:

1. **Initialization**:
    - The algorithm begins by initializing basic variables including **b**, **beta**, and **alpha**, with values ranging between 0 and 1. In the proposed algorithm, these variables are initially set to 0.5. During each collision event, **beta** and **alpha** are updated with random values within this range, while **b** changes in each generation.
    - Additionally, the algorithm accepts an input mutant file from the file system, which contains the initial mutants to be processed.
2. **Handling Single Mutants**:
    - In cases where a module contains only one mutant, inter-molecular collisions (synthesis and ineffective collisions) cannot be applied. Therefore, the algorithm checks the number of mutants within each module using a **mutant_length** variable.
    - When **mutant_length** equals 1, indicating a single mutant, the algorithm resorts to uni-molecular collision methods. Specifically, it first applies the **onWallineffective()** collision to enhance the mutant's performance, followed by decomposition to create additional mutants.
3. **Collision Selection**:
    - For modules with more than one mutant, the algorithm determines the type of collision to apply based on the value of **b**. If an inter-molecular collision is selected, the **alpha** value is used to decide between synthesis or an inter-molecular ineffective collision.
    - Conversely, if a uni-molecular collision is chosen, the **beta** value determines whether to apply decomposition or an on-wall ineffective collision.
4. **Iteration and Termination**:
    - The algorithm repeats the above processes iteratively until the stopping condition, defined by the user, is met. Each iteration updates the state of the mutants according to the collision types and parameters specified.
    - Upon reaching the stopping condition, the algorithm terminates and outputs the final solution.

The proposed algorithm is presented below

Proposed Algorithm

Input:

- Initial mutants file
- Parameters: genSize (maximum iterations), b (collision type selector), alpha (inter-molecular collision selector), beta (uni-molecular collision selector)

Output:

- Optimized set of Higher Order Mutants (HOMs) in XML format

Initialization Phase

- Set initial values for parameters:
  - b = 0.5, alpha = 0.5, beta = 0.5
- Load initial mutants from the file system.
- Set stopping condition genSize.

Iteration Phase

- While stopping condition (genSize) is not met:
  - For each module, determine mutant_length.
    - If mutant_length == 1:
      - Apply onWallineffective() collision to the single mutant.
      - Apply Decomposition to generate additional mutants.
    - Else (mutant_length > 1):
      - Generate a random value for b.
      - If b > 0.5:
        - Generate a random value for alpha.
        - If alpha > 0.5, perform Synthesis to merge two mutants.
        - Else, perform Inter-Molecular Ineffective Collision using crossover operations.
      - If b ≤ 0.5:
        - Generate a random value for beta.
        - If beta > 0.5, perform Decomposition on the mutant with the highest hit count.
        - Else, perform On-Wall Ineffective Collision on a low-fitness mutant.
  - Update mutants' states based on collision outcomes.
  - Increment iteration count.

Finalization Phase

- If stopping condition is met:
  - Save final set of mutants in XML format.
  - Terminate the algorithm.

End Algorithm

4. Implementation of the Proposed Algorithm

The proposed algorithm, named HOMUsingCRO (Higher-Order Mutant Using Chemical Reaction Optimization), is implemented using Java. This model is designed for higher-order mutant generation by applying the CRO algorithm. The implementation is intended to facilitate Java program testing. The main features of the HOMUsingCRO interface are described below:

Figure 2 illustrates the layout of the HOMUsingCRO application, which is divided into three primary sections:

1. **Directory Selection for Mutants**:
    - **Upper Section**: This section contains the first text field and a "Browse" button. The "Browse" button allows users to select a directory containing the first-order mutant files. Upon clicking this button, a file dialog opens, restricting the user to choosing directories only. The selected directory path is then displayed in the adjacent text field.
2. **Directory Selection for Test Cases**:
    - **Middle Section**: This section features a second "Browse" button and a corresponding text field. The second "Browse" button is used to select the directory containing the test case files. Similar to the previous dialog, this button opens an open dialog box that only allows directory selection. The path to the selected test case directory is displayed in the adjacent text field.
3. **Execution Status and Control**:
    - **Last Section**: This section includes a text area that provides feedback on the algorithm's execution status. A success message is displayed in the text area once the algorithm has completed its execution.
    - **Run CRO Button**: This button initiates the execution of the CRO algorithm. When clicked, it starts the process of mutant generation and optimization as per the CRO methodology.

The user interface of HOMUsingCRO is designed to be intuitive, allowing users to easily select directories for mutants and test cases, and to monitor the status of the algorithm. The system ensures smooth execution and clear communication of the results.

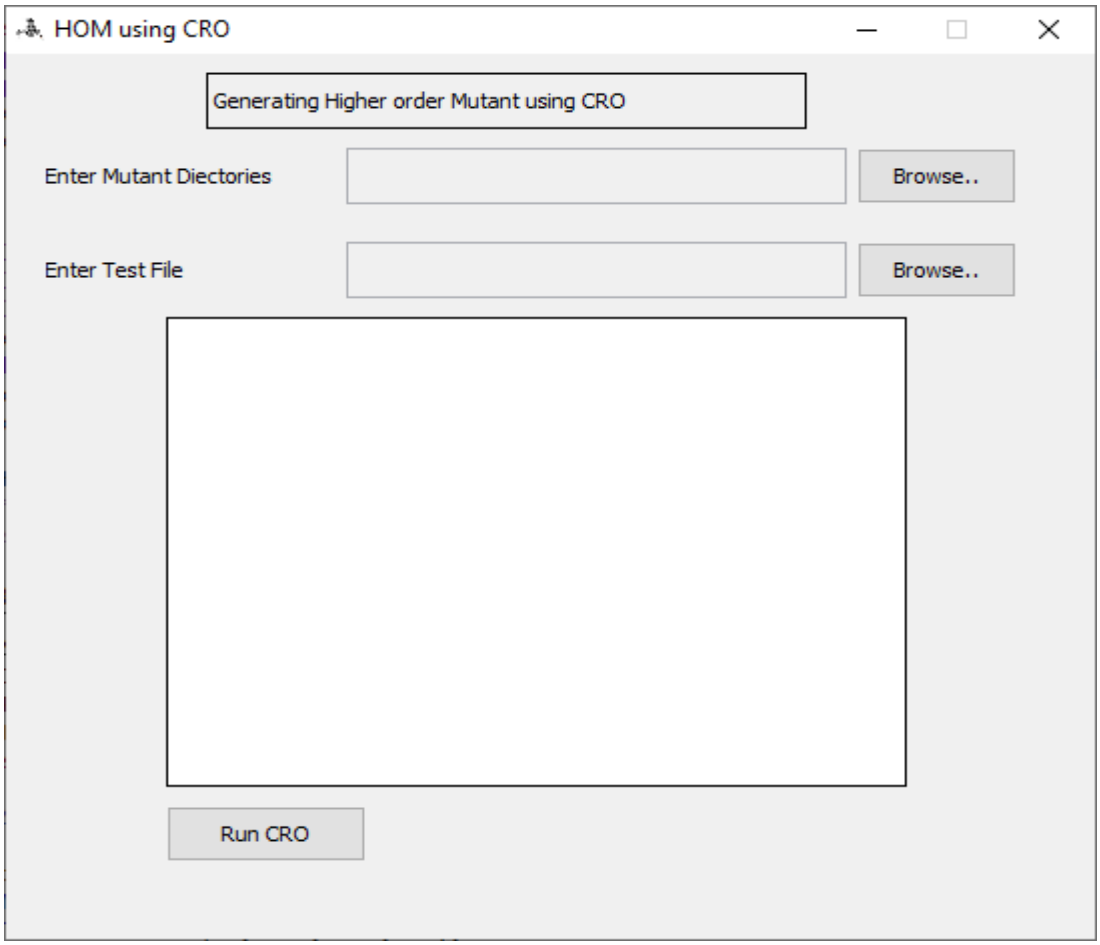

Fig. 2. HOMUsingCRO graphical user interface

4.1 Subject Program

For the experimental evaluation of the proposed algorithm, a Java project named **Store Management System** was utilized. This project serves as a student graduation project and encompasses a range of functionalities relevant to basic store operations. The key features of the Store Management System include item registration, post-sale item reduction, daily profit calculation, inventory counting, and other essential store activities.

The Store Management System comprises two Java class files:

1. **Abstract Class**: The first file is an abstract Java class that contains a total of 38 methods. This class defines the foundational methods and structure necessary for the system's operations.
2. **Concrete Implementation Class**: The second file extends the abstract class and provides concrete implementations for all 38 methods. This class is composed of 607

lines of code, integrating the abstract methods with functional code to perform the system's operations.

Table 2 : Subject Programs

Project Name	Number of Class	Line of code	Number of methods
Store management system	2	754	56

This subject program was chosen to evaluate the effectiveness of the HOMUsingCRO algorithm in generating higher-order mutants and optimizing the software testing process.

4.2 Tools to generate initial Mutant MuJava

MuJava is an open-source tool designed for mutant generation in Java programs. It is available online and provides a comprehensive suite of resources, including installation guides and references to other tools necessary for its operation. MuJava facilitates the automatic creation of first-order mutants, enabling researchers and testers to execute test suites, analyze the results, and improve software testing processes. The tool supports both traditional and object-oriented programming paradigms. The tool is divided into three main components:

4.3 Mutant Generator

The **Mutant Generator** is a core component of MuJava, designed to produce mutants for both traditional and object-oriented programming. It achieves this by applying operators at both the traditional level and the class level. You can see the user interface of the mutant generator in Figure 3.

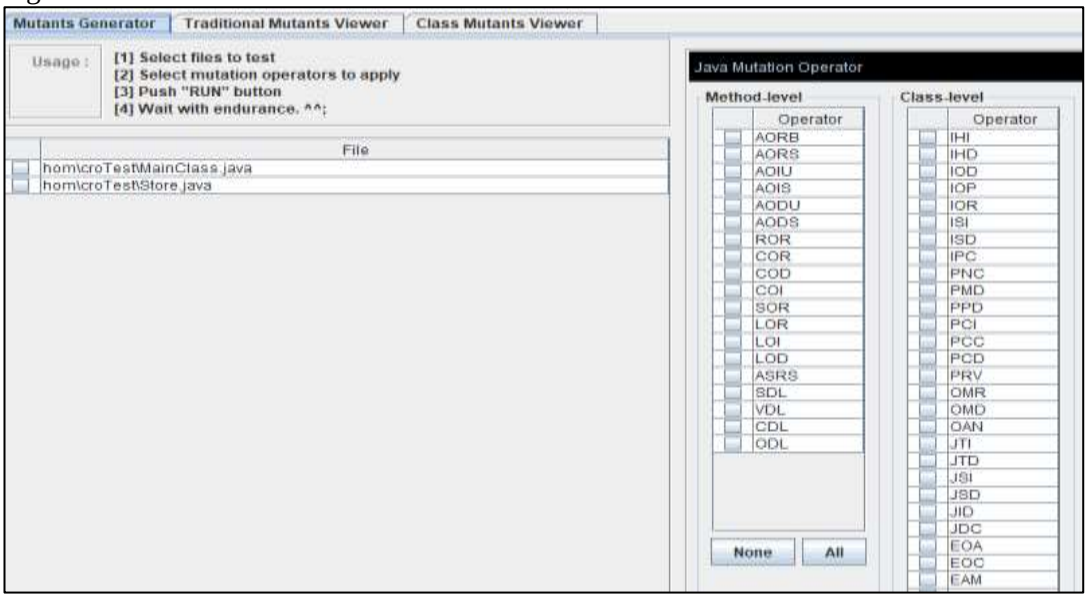

Fig. 3. Mutant generator user interface

4.4 Mutant viewer

The **Mutant Viewer** is a component of MuJava that provides users with a detailed view of the generated mutants and the modifications they introduce to the source code. It plays a crucial role in helping users understand the impact of each mutant on the original codebases. The following figure shows the mutant viewer user interface (Fig.4).

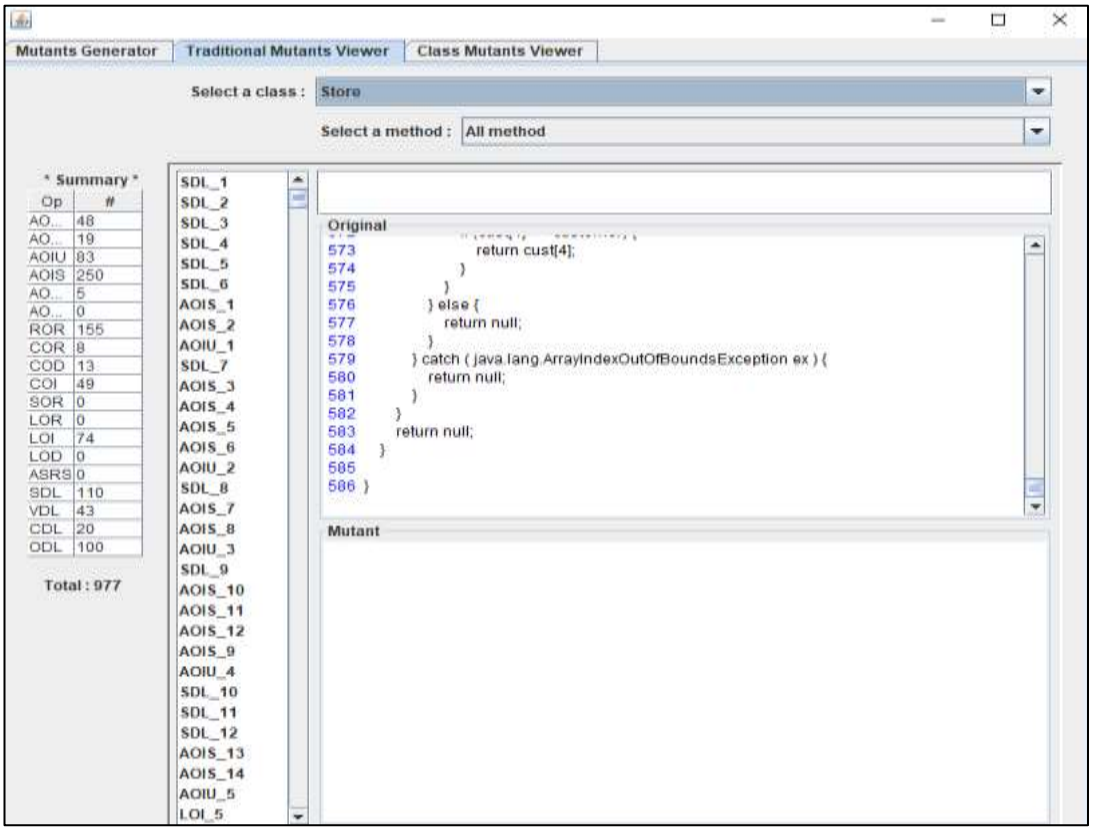

Fig. 4. Generated Mutant

4.5 Test case generation process

For the test case generation, this research utilized **JUnit 5**, a widely recognized framework for writing and executing tests in Java. JUnit 5 enables the creation of test cases to evaluate Java programs effectively. In the context of this study, the Store Management System, which consists of 14 main modules, required a comprehensive set of test cases. Accordingly, 14 distinct test suites were developed, each corresponding to one of the modules. Each test suite includes 10 test cases, resulting in a total of 140 test cases across all suites.

Each test class within JUnit 5 is equipped with two critical methods to manage the testing lifecycle. The **setUpBeforeClass()** method, annotated with **@BeforeClass**, is executed once

before any of the test methods are called. This method is used to initialize the objects of the mutant class, ensuring they are prepared for testing. Conversely, the **tearDownAfterClass()** method, annotated with **@AfterClass**, is invoked after all test methods have completed. It is responsible for cleaning up by setting the class object under test to an empty or null state. Additionally, each test method is marked with the @Test annotation to indicate that it is a test case.

The HOMUsingCRO tool, which is used in conjunction with JUnit 5, provides an interface for selecting directories that contain mutant files and source code. Users can browse and select these directories using **JFileChooser**, with the selected file being assigned to a variable. The **listFiles()** method is then employed to retrieve subdirectories from the chosen directory. Each file is read line by line to identify the method affected by the mutant.

MuJava, the tool used for generating mutants, organizes mutant files in directories named by combining the return type and method name. For instance, if the original method is **int sum(int a, int b)**, MuJava generates a directory named **int_sum(int, int)** to store mutants related to that method, as illustrated in **Figure 5**. This structured approach ensures that the generated test cases are comprehensive and capable of effectively detecting faults introduced by the mutants.

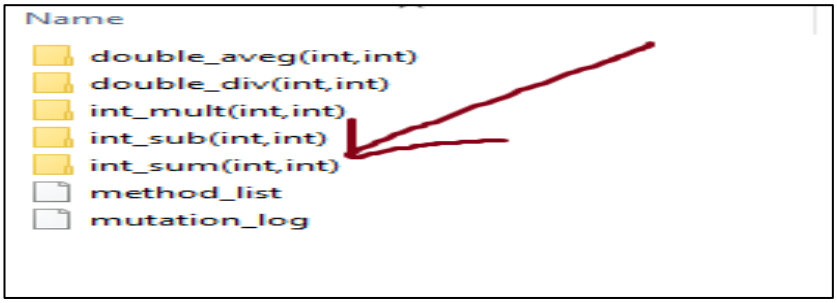

Fig. 5. muJavacreates a mutant directory by method name and return type

To identify the method associated with a mutant from the mutant source code, the process involves using a substring to match key elements such as **int** and **sum**, which correspond to the method's signature. The proposed method stores each file in a **LinkedList** variable, a dynamic array that accommodates variable sizes. Once the method where the mutant is located is identified, the system determines the line number where the method's ending brace is positioned by employing a stack data structure. This technique allows the program to ascertain the start and end lines of the method in the source code.

The proposed model employs a **while** loop to read both the original and mutant source codes line by line. As the loop progresses, it checks whether the current line pertains to the method containing the mutant. Upon reaching the method, an inner **while** loop is initiated, which continues to iterate until the end of the method is reached. During this iteration, the inner loop compares each line of the mutant code with the corresponding line of the original code in a

parallel fashion. Any discrepancies between the lines are recorded as differences, which represent the mutant code.

For accurate comparison, both the mutant and original source codes must be properly formatted, as shown in **Figure 6**. This structured approach ensures that the mutant code is effectively identified and differentiated from the original code, facilitating a thorough analysis of the mutants within the source code.

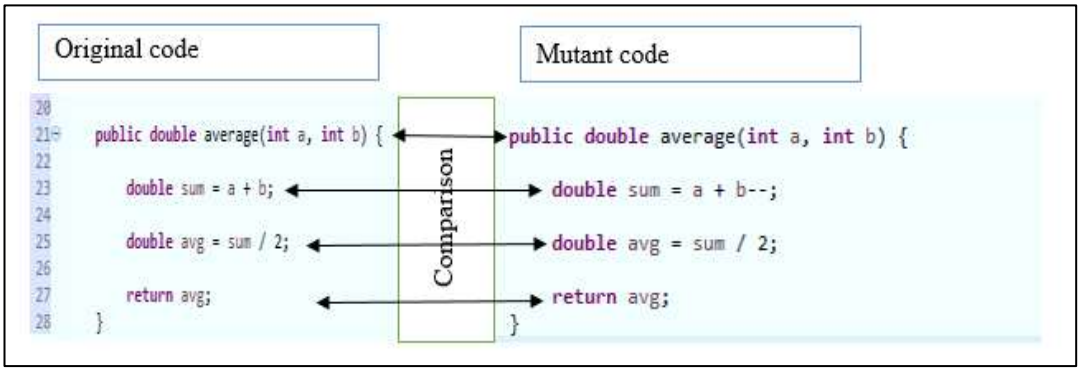

Fig. 6. Mutant and original code comparison

The program performs a line-by-line comparison of the code from both the original and mutant source files. During this comparison, the program checks each line to identify discrepancies between the two versions. For instance, as illustrated by the provided code snippet, lines 21, 25, and 27 in both files are identical. However, a difference is observed in line 23, where the statement in the mutant code diverges from the statement in the original code. This discrepancy is crucial as it highlights the mutant statement.

The mutant statement found in line 23 is recorded as a mutant and subsequently stored in the **LinkedList** variable. This collection of mutants is then subjected to further processing to analyze the impact and effectiveness of the introduced changes. By systematically identifying and cataloging these differences, the program ensures that each mutant is properly documented and can be used for subsequent testing and evaluation.

4.6 How to Remove Irrelevant Mutants

As detailed in the previous section, the program identifies mutant code by traversing directories within the same module. When a mutant statement is discovered, the line number where this mutant occurs is recorded in the **LinkedList** variable. During subsequent iterations, the program performs a check to determine if the line number of the newly found mutant is already present in the **LinkedList**.

If the line number is found in the **LinkedList**, the program recognizes this as a repeated mutant and omits it from further processing. This step ensures that only unique mutants are retained, avoiding redundancy. Conversely, if the line number does not exist in the **LinkedList**, the program adds the mutant file to the **LinkedList** for further operations. This approach effectively filters out irrelevant or duplicate mutants, streamlining the process of mutant analysis and maintaining the focus on novel mutants that contribute to meaningful testing and evaluation.

Algorithm to identify and remove repeated mutant

Step 1: Read an array of mutant files and original files as input.

Step 2: Read each muted source code and source code line by line and move to step 3 until the line number checker reaches null.

Step 3: Identify the method name where the mutant is located and also identify the method ending and starting brace of the method where the mutant is located and then move to step 4.

Step 4: Identify the line number where the muted statement is located and store the line number and mutant file in the linked list variable.

Step 5: If the current muted statement line number is not equal to the method ending brace line number, then go to step 6, else move to step 2.

Step 6: If the statement read from the muted file and the statement read from the original file are different, pass the line number to step 7 else move to step 2.

Step 7: Read the mutant line number from step 6 and check if the line number exists in the mutant list, then consider the mutant as a repeated mutant, skip storing the mutant-to-mutant list, and go to step 2, else consider the mutant as a relevant mutant and store the mutant in mutant linked list variable for farther operation.

Step 8: Repeat step 3 till the line number checker returns null.

4.7 Process of Running Test Cases Over Mutants

The execution of test cases is a critical component of the HOMUsingCRO architecture and is invoked during each generation execution. This process comprises three main sequential components: mutant selection and object creation, test runner, and XML report generation. Each of these components plays a vital role in executing a mutant against the test cases and generating the corresponding reports.

4.7.1 Mutant Selection and Object Creation

The process begins with the **runTestOverHOM()** method, which is defined in the test execution class. This method initiates the test execution by first browsing the file system to access the mutant files. Each mutant class file, along with its module name, is read using a

for-each loop. During each iteration of the loop, the selected mutant and its module name are passed to the **setClassObject()** method, which is responsible for creating a mutant object. Since the mutant class file is initially outside the Java classpath, it is relocated to the appropriate classpath to facilitate execution. After moving the file, the tool compiles the mutant class located in the classpath. Using the **Class.forName()** method, the tool creates an instance of the mutant class with **newInstance()**, assigning it to a variable of the same type as the new class instance. Finally, the **testRunner()** method is invoked with the module name and mutant file as parameters to commence the test execution.

4.7.2 Test Runner

The **testRunner()** method is designed to handle the execution of mutants within a module. Each test suite is saved under the respective module's name, aiding in the selection of the appropriate test suite for each mutant. The method iterates over all test suite files from the file system using a **foreach** loop and selects the correct test suite by comparing the test suite file name with the mutant module name. If the names match, the corresponding test suite is chosen for execution. The tool then creates a test suite object using **Class.forName()** and passes this object to **JUnitCore.runClasses(),** which executes the selected test suite class. The method returns a Result object, which is used to gather information on failed tests and the total number of tests run for the mutant. Subsequently, the **HOMXMLReport()** method is called with parameters including the number of killed tests, total tests, module name, and fitness value to generate a detailed XML report.

4.7.3 XML Report Generator

The **HOMXMLReport()** method is responsible for creating an XML-based report for each mutant class file. Initially, an XML file named HOMUsingCRO.xml is created to store data for all mutants. Using a Java XML parser, the method constructs XML elements to record details such as killed mutants, fitness value (or kinetic energy), and the total number of mutants. The XML report encompasses information on total killed mutants, total generated mutants, mutation scores, number of hits, and kinetic energy for each mutant. An example of the XML report format is illustrated in **Figure 7**.

Fig. 7. Sample XML Report

4.8 The architecture of the proposed tool HOMUsingCRO

The tool developed using the proposed algorithm is named HOMUsingCRO. Its architecture is structured into three main layers: the Upper Layer, the Middle Layer, and the Lower Layer. Each layer has distinct components that contribute to the overall functionality of the tool.

4.8.1 Upper Layer

The Upper Layer represents the graphical user interface (GUI) of HOMUsingCRO. It includes the following elements:

- **Buttons**: There are three buttons:
  - Two buttons are used for browsing directories: one for the mutant directory and one for the test suite directory.
  - The third button initiates the execution process of the algorithm.
- **Text Fields**: Two text fields display the paths of the selected directories for mutants and test suites.
- **Text Area**: This area provides feedback by displaying a message indicating the completion of the algorithm's execution.

4.8.2 Middle Layer

The Middle Layer encompasses the core functionalities of the tool, which include:

- **Test Runner**: This component executes JUnit test cases on the selected mutants and records the number of failed tests and the total number of tests executed.
- **Relevant Mutant Selector**: This component filters out redundant mutants by removing those deemed irrelevant.
- **XML Report Generator**: After test execution, this component stores the results in an XML format on the file system.
- **File Management**: This component handles reading mutant and test files from the file system and writing higher-order mutants back to the file system.
- **Mutant Selector for CRO Operator**: This component selects mutants for processing by the proposed Chemical Reaction Optimization (CRO) operator based on predefined conditions.
- **CRO Algorithm**: This component implements the four operators defined in the CRO algorithm.
- **Test Case Selection**: This component facilitates the selection of appropriate test cases for each mutant and passes them to the Test Runner.
- **Source Code Compiler**: This component compiles the mutant and test case source codes after moving them to the correct Java classpath.

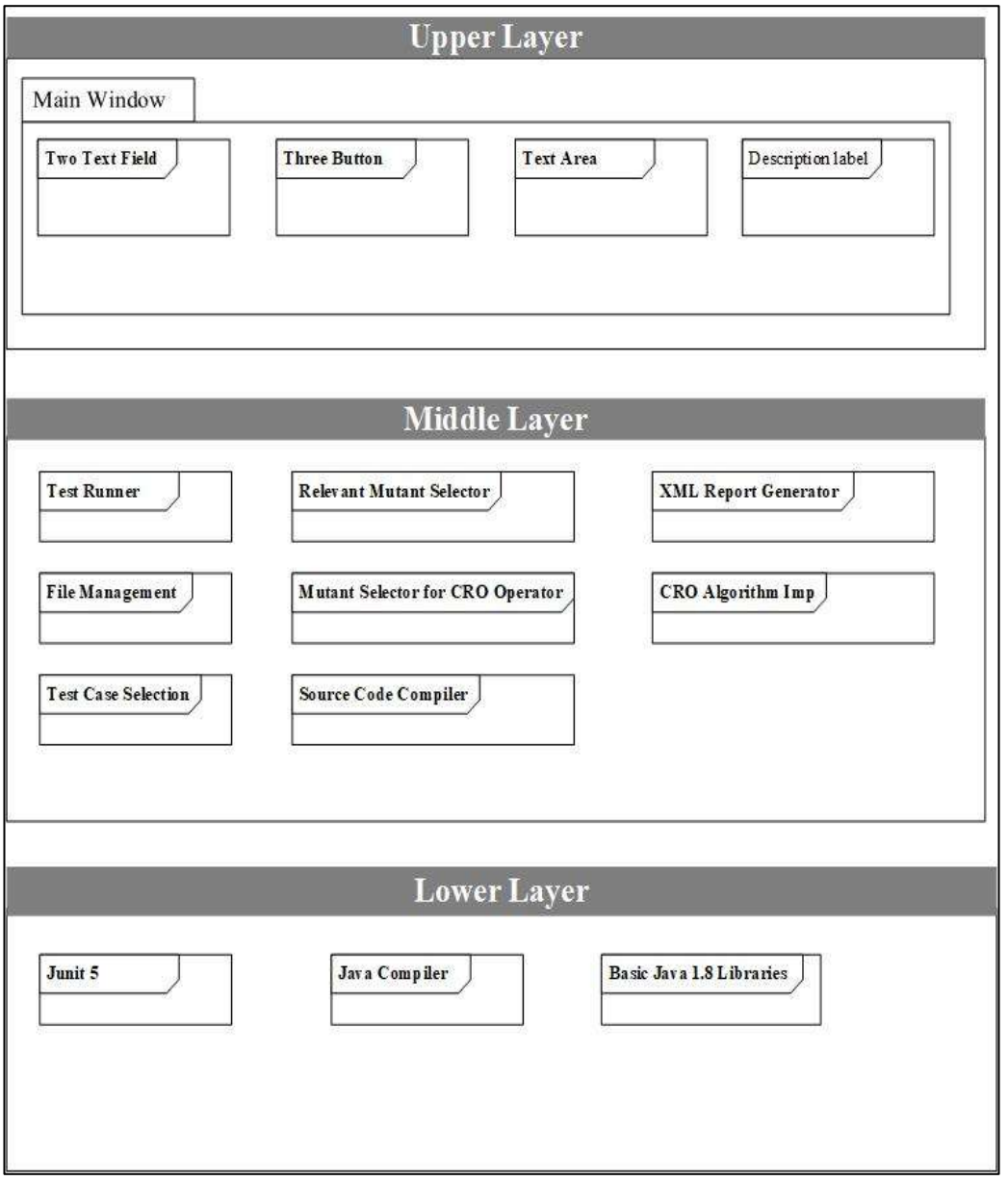

Fig. 8. The architecture of the proposed approach

4.8.3 Lower Layer

The Lower Layer contains external Java libraries essential for the tool's operation:

- **JUnit**: A Java-based testing framework used for writing and running JUnit test programs.

- **Java Compiler**: Part of the common Java libraries, this tool helps compile Java source code programmatically.
- **Java 1.8 Libraries**: Includes common I/O libraries, Swing libraries for GUI development, and other standard Java libraries.

4.9 Execution Process of HOMUsingCRO

The execution process of HOMUsingCRO involves several key steps to ensure that the tool efficiently processes mutants and generates results. Here's a detailed outline of the process:

1. **Input Preparation**: The tool requires two inputs: a mutant directory and a test suite directory. The mutant directory contains subdirectories, with the final subdirectory holding the mutant source code and byte code files (.java and .class). The test suite directory provides the test cases. The tool reads the mutant directory and filters out relevant mutants.
2. **Classpath Adjustment**: Since the mutants and test suite files are initially not in the correct Java classpath, the tool moves these files to the appropriate classpath.
3. **Compilation**: The tool compiles both the mutant files and the test suite files that have been moved to the correct Java classpath.
4. **Test Execution**: The test suite is run against the selected mutant. During this phase, the fitness value and mutation score for each mutant are calculated based on the test results.
5. **CRO Operator Selection**: Based on the calculated fitness values and other parameters, the appropriate Chemical Reaction Optimization (CRO) operator is selected. The tool then selects a mutant for the chosen CRO operator and applies the CRO algorithm.
6. **Mutant Generation**: The application of the CRO algorithm results in the generation of new mutants. These newly generated mutants are then compiled.
7. **Re-Execution and Recalculation**: The test suite is run again against the newly generated mutants. The fitness values and mutation scores for these new mutants are recalculated.
8. **Iteration and Termination**: If the stopping condition is not met, the process loops back to the CRO operator selection step. If the stopping condition is met, the execution terminates, and the final output is generated.

This process ensures a systematic approach to mutant processing, including compilation, testing, optimization, and re-testing, culminating in the generation of comprehensive results.

5. Results and Discussion

5.1 CRO Algorithm Setup

Before initiating the execution of the Chemical Reaction Optimization (CRO) algorithm, the user specifies the number of iterations for the algorithm. This flexibility allows the algorithm

to be adjusted according to the specific requirements of the experiment, thereby providing insights into how iteration counts affect mutant generation and mutation scores.

For this study, the CRO algorithm was configured to run with varying numbers of generations: 10, 30, and 50. In each generation, the entire set of mutant modules was executed, and one CRO elementary reaction was applied to each mutant module. The experiment involved processing 1,170 first-order mutants (FOM) and 140 test suite files generated by JUnit. After applying the CRO mutant filtering technique, 420 mutants were identified as relevant. To ensure proper execution, the test suite files and mutant files were relocated to the correct Java classpath. This step was crucial, as files need to be in the correct classpath to be active for execution. The experiments were conducted on an HP laptop equipped with an Intel Core i5 processor and 4GB of RAM. This setup was used to evaluate the performance and effectiveness of the CRO algorithm in generating and filtering mutants.

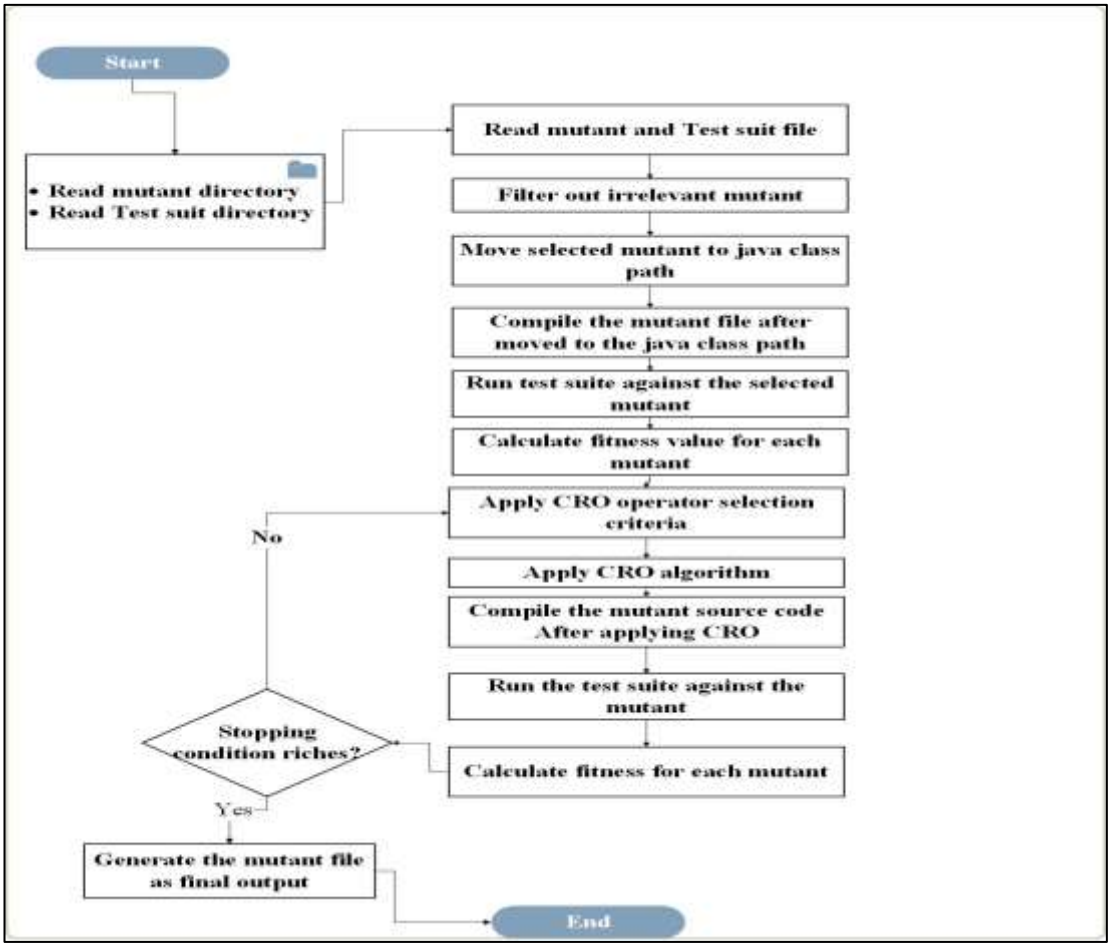

Fig. 9. Execution process of HOMUsingCRO

5.2 Test case execution result before applying CRO

Once the tool accepted the mutant files from the file system, it proceeded with filtering out the relevant mutants. These mutants were then relocated to the project directory to streamline the processing. To facilitate organization and clarity, the mutants were renamed with a directory prefix "CRO_XXX," where "XXX" represents the specific module name or method name from the program. This renaming process is illustrated in Figure 10. The directory naming convention helps in managing and tracking the mutants more effectively during the subsequent stages of testing and analysis.

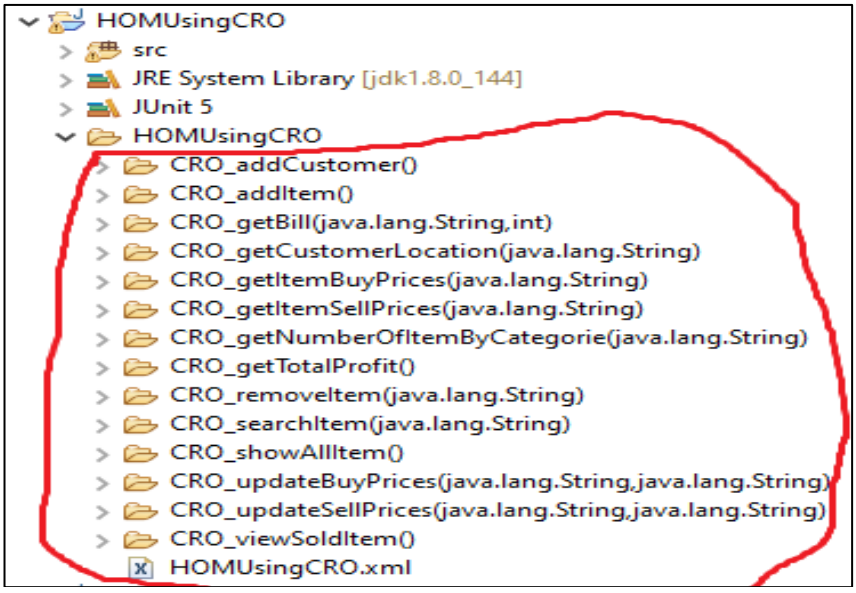

Fig. 10. Generated Mutant File Structure

HOMUsingCRO accepts test suite files from the file system and transfers them to the project's src directory inside the test package. Subsequently, the test suite is executed against the mutant files. The tool processes each mutant module individually, selecting the appropriate test suite file based on a match between the mutant module name and the test suite name.

Each test suite consists of 10 test cases, with a default distribution of 50% failed tests and 50% passed tests. After executing the test suites, it was observed that all mutants were killed by at least one test case, resulting in a mutation score of 100%. This outcome indicates that all mutants were effectively identified by the test suite, suggesting that the test suite has a high capability to detect errors introduced by the mutations. Figure 11 provides a visual representation of the test case execution results before the application of the CRO algorithm.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HOM-Report>

<killed-mutant>420.0</killed-mutant>
<total-mutant>420.0</total-mutant>
<mutation-score>100.0%</mutation-score>

========================================
```

Fig. 11. Mutant Execution Result before CRO

5.3 Execution process and its outcome

The number of iterations executed by the algorithm is user-defined. To assess the impact of the number of generations on the mutant creation process, the experiment specifically evaluates three different generation counts: 10, 30, and 50. For each generation count, the algorithm compares the mutation score value, the number of generated mutants, and the number of killed mutants with the initial values.

Initially, all 420 mutants were effectively killed by the given test cases, resulting in a mutation score of 100%. When the number of generations was set to 10, the algorithm performed four elementary reactions of CRO, selected randomly. After 10 iterations, the results showed a total of 416 mutants generated, a 1% decrease from the initial count. Out of these, 408 mutants were killed, which is a 3% decrease from the initial number of killed mutants. Consequently, the mutation score dropped to 98%, a 2% reduction from the initial score. The execution of the algorithm with 10 generations took approximately 45 minutes.

Given that these results were not satisfactory, an additional iteration with a higher number of generations was conducted. Figure 12 illustrates the execution results for 10 generations.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HOM-Report>

<killed-mutant>408.0</killed-mutant>
<total-mutant>416</total-mutant>
<mutation-score>98.07%</mutation-score>

========================================
<module name="addCustomer">
<parent ke="0.5" name="CRO_de_1651" num-hit="1"/>
<parent ke="0.5" name="CRO_de_419" num-hit="1"/>
<parent ke="0.5" name="CRO_de_6" num-hit="1"/>
<parent ke="0.5" name="CRO_de_720" num-hit="1"/>
<parent ke="0.5" name="CRO_de_771" num-hit="1"/>
<parent ke="0.5" name="CRO_In_1568" num-hit="1"/>
<parent ke="0.5" name="CRO_Sy_2253" num-hit="2"/>
</module>========================================

<module name="addItem">
<parent ke="0.5" name="CRO_In_1579" num-hit="0"/>
<parent ke="0.5" name="CRO_In_1766" num-hit="1"/>
<parent ke="0.5" name="CRO_On_438" num-hit="4"/>
<parent ke="0.5" name="CRO_Sy_1096" num-hit="1"/>
<parent ke="0.5" name="CRO_Sy_1800" num-hit="2"/>
</module>
```

Fig. 12. Execution result when generation equal to 10

When the number of generations was set to 30, the experiment yielded 320 mutants, representing a 23% reduction from the initial 420 mutants. Among these, 209 mutants were killed by the test cases, which is a 50% decrease compared to the initial number of killed mutants. This result indicates that the mutants generated with 30 iterations were more challenging to detect than both the initial mutants and those generated with 10 iterations. The mutation score dropped to 65%, reflecting a 35% decrease from the initial score. The algorithm required 110 minutes to complete the execution for 30 generations. Overall, this iteration provided a more effective mutation score, a lower number of generated mutants, and a higher proportion of killed mutants compared to the initial results and the results from 10 generations. Figure 13 illustrates the execution outcome when the generation count was 30.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HOM-Report>

<killed-mutant>209.0</killed-mutant>
<total-mutant>320.0</total-mutant>
<mutation-score>65.3125%</mutation-score>

====================================
<module name="addCustomer">
<parent ke="0.5" name="CRO_de_700" num-hit="1"/>
<parent ke="0.5" name="CRO_In_1421" num-hit="5"/>
<parent ke="0.5" name="CRO_In_2111" num-hit="4"/>
<parent ke="0.5" name="CRO_In_714" num-hit="3"/>
<parent ke="0.5" name="CRO_On_1007" num-hit="5"/>
</module>====================================

<module name="addItem">
<parent ke="0.5" name="CRO_de_1593" num-hit="0"/>
<parent ke="0.5" name="CRO_de_1707" num-hit="1"/>
<parent ke="0.5" name="CRO_de_1919" num-hit="1"/>
<parent ke="0.5" name="CRO_de_194" num-hit="1"/>
<parent ke="0.5" name="CRO_de_244" num-hit="1"/>
<parent ke="0.5" name="CRO_de_578" num-hit="1"/>
<parent ke="0.5" name="CRO_de_61" num-hit="1"/>
<parent ke="0.5" name="CRO_de_774" num-hit="1"/>
<parent ke="0.5" name="CRO_In_1016" num-hit="3"/>
<parent ke="0.5" name="CRO_On_112" num-hit="4"/>
<parent ke="0.5" name="CRO_Sy_1684" num-hit="2"/>
</module>
```

Fig. 13. Execution result when generation equal to 30

When the number of generations was set to 50, the experiment produced a total of 108 mutants, a 75% reduction from the initial 420 mutants. Of these, only 30 mutants were killed by the test suite, which represents a 93% decrease from the initial number of killed mutants. Despite the significant drop in the number of killed mutants, the mutation score improved to 22.77%, which is a 72% increase compared to the original mutation score. The execution process took 216 minutes to complete for 50 generations. These results suggest that increasing the number of iterations can lead to more refined mutant generation and improved mutation score outcomes. Figure 14 illustrates the execution results for 50 generations.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HOM-Report>

<killed-mutant>31.0</killed-mutant>
<total-mutant>113.0</total-mutant>
<mutation-score>27.43362831858407%</mutation-score>

=====================================
<module name="addCustomer">
<parent ke="0.5" name="CRO_de_1036" num-hit="5"/>
<parent ke="0.5" name="CRO_de_1440" num-hit="4"/>
<parent ke="0.5" name="CRO_de_1578" num-hit="2"/>
<parent ke="0.5" name="CRO_de_1781" num-hit="2"/>
<parent ke="0.5" name="CRO_de_1865" num-hit="2"/>
<parent ke="0.5" name="CRO_In_1974" num-hit="3"/>
</module>=====================================

<module name="addItem">
<parent ke="0.5" name="CRO_de_1433" num-hit="2"/>
<parent ke="0.5" name="CRO_de_1643" num-hit="1"/>
<parent ke="0.5" name="CRO_de_1887" num-hit="3"/>
<parent ke="0.5" name="CRO_de_1977" num-hit="2"/>
<parent ke="0.5" name="CRO_de_516" num-hit="4"/>
<parent ke="0.5" name="CRO_de_835" num-hit="1"/>
<parent ke="0.5" name="CRO_In_1102" num-hit="5"/>
<parent ke="0.5" name="CRO_In_250" num-hit="4"/>
</module>
=====================================
```

Fig. 14. Execution result when generation equal to 50

5.4 Mutants before and after proposed algorithm

This research utilized MuJava to generate initial mutants for the algorithm by applying it to the Store Management System project. Initially, a total of 1170 first-order mutants were generated, comprising 896 mutants created using 19 traditional-level operators and 274 mutants created using 28 class-level operators.

Upon applying the proposed algorithm filtering technique, the number of relevant mutants was reduced to 420. Each of these mutants was accompanied by an XML file detailing the method name, the line number where the mutated statement exists, the method's opening and ending braces, and the minimum hit number. This filtering process also involved removing all .class files to reduce execution time and creating new mutants based on the properties of the old ones. The application of CRO elementary reactions resulted in changes to the mutation score and other properties of the mutants.

The following table summarizes the mutant properties before and after applying the CRO technique:

Table 3: Mutant result before and after Proposed algorithm HOMUsingCRO

Properties	**Before Applying the HOMUsing CRO Filtering Technique**	**After Filtering the HOMUsing CRO Filtering Technique**	**After Applying HOMUsing CRO Gen=10**	**After Applying HOMUsing CRO Gen=30**	**After Applying HOMUsing CRO Gen=50**
No. of input mutants	1170	420	420	420	420
Generated Mutants	-	-	416	320	108
No. of killed mutants	1170	420	408	209	30
No. of test cases	140	140	140	140	140
Mutation score	100%	100%	98.07%	65.31%	27.77%
Execution time	5 minutes	1 minute	46 minutes	110 minutes	216 minutes

5.5 The proposed algorithm Vs another algorithm on the process of higher-order mutant generation

In this section, we compare our study with three previous studies, focusing on subject programs, problems addressed, techniques employed, and the strengths and limitations relative to our proposed method.

When compared to Anas et al. [2], our study utilizes the Store Management project, which comprises 607 lines of code, 38 methods, and two classes. In contrast, Anas et al. used a smaller project with just one class, 23 methods, and 315 lines of code. This indicates that our subject program is larger and more complex, providing a more challenging environment for mutant generation and testing. In terms of mutant and test case ratios, our study involved 420 mutants and 140 test cases, resulting in a ratio of 3:1. Anas et al., on the other hand, worked with 2000 mutants and 800 test cases, yielding a ratio of 5:2. This comparison highlights a more balanced ratio in our approach. While Anas et al. applied a genetic algorithm and achieved a mutation score of 99%, their method did not reduce the number of higher-order mutants (HOMs), maintaining the initial count of 2000 mutants. Conversely, our CRO-based method improved the mutation score by 73% and reduced the number of mutants by 73%. This demonstrates that our approach not only effectively reduces mutants but also maintains high

mutation scores. The potential exists for our method to achieve similar high mutation scores as Anas et al. with increased iterations.

In comparison with Nguyen et al. [1], our study initially generated 1170 first-order mutants (FOMs), which were reduced to 420 FOMs using CRO filtering. After 50 iterations, we produced 108 higher-order mutants (HOMs), marking a 25% reduction from the filtered FOMs. Nguyen et al., on the other hand, employed a large subject program consisting of five Java projects, each with 51 to 144 class files. This broader testing context might influence the effectiveness of their algorithm. Our approach achieved a 91% reduction in the total number of mutants and a mutation score of 27% with the generated HOMs, demonstrating strong performance in realistic fault generation. While Nguyen et al.'s approach was tested on a larger scale, it did not match the reduction effectiveness observed with our method(Table 4).

Table 4: Comparison of Proposed algorithm HOMUsingCRO with other existing algorithm

Criteria	Proposed Algorithm	Anas et al. [2]	Nguyen et al. [1]
Subject Program	Store Management Project (607 LOC, 38 methods, 2 classes)	Smaller project (315 LOC, 23 methods, 1 class)	Large-scale program (5 Java projects, 51-144 class files per project)
Mutants Generated	1170 FOMs (reduced to 420 FOMs using CRO filtering, 108 HOMs after 50 iterations)	2000 mutants (no reduction of HOMs)	Not specified, focus on large-scale testing context
Test Cases	140 test cases	800 test cases	Not specified
Mutant/Test Case Ratio	420 mutants : 140 test cases (3:1)	2000 mutants : 800 test cases (5:2)	Not specified
Mutation Score	27% after 50 iterations (indicating hard-to-kill HOMs, aligned with the objective of generating complex mutants)	99%	Not specified
Mutant Reduction	91% reduction in total mutants (73% improvement in mutation score)	No reduction in HOMs (2000 mutants retained)	Less effective in mutant reduction

Strengths	Generates hard-to-kill HOMs, effectively reduces mutants, and encourages more robust test suite creation	High mutation score (99%), but no focus on HOM complexity	Broader context due to large subject programs
Limitations	Lower mutation score (27%) as HOMs are harder to kill, but this is a positive outcome in terms of the objective of generating complex mutants	Did not address mutant reduction, no HOM reduction	Did not achieve the same level of mutant reduction as our method

HOMUsingCRO effectively balances mutant reduction and mutation score improvement, outperforming previous methods in both areas. Anas et al.'s approach achieved high mutation scores but did not address mutant reduction, while Nguyen et al.'s larger subject programs provided a broader context but did not achieve the same level of mutant reduction. Overall, our proposed algorithm offers a more effective solution for managing and evaluating mutants.

6. Conclusion

This study presented an innovative approach to generating higher-order mutants (HOMs) using the Chemical Reaction Optimization (CRO) algorithm. The method effectively balances the reduction in the number of mutants while increasing their complexity, making them more challenging to detect. By applying the CRO algorithm's four operators on-wall ineffective, synthesis, decomposition, and inter-molecular ineffective this approach improves the rigor of mutation testing, helping to address the limitations in existing techniques that often prioritize either mutant reduction or complexity but not both.

The experimental results demonstrated that with increased iterations (up to 50), there was a significant reduction (93%) in the total number of mutants while increasing their resistance to detection, as evidenced by a lowered mutation score (27.77%). This study's findings underscore the CRO algorithm's capability to improve mutation testing in more complex and realistic scenarios. When compared to previous approaches, this method showed better performance in reducing the number of mutants and maintaining a high mutation score, making it a valuable tool for enhancing software test suite evaluation.

Future research should explore the application of the CRO algorithm across larger and more diverse subject programs, including languages like C++ and C#, to validate its effectiveness in different domains. Additionally, optimizing the algorithm to handle even larger scales of mutation testing will further contribute to improving fault detection in software systems.

References

[1] Nguyen, Quang Vu, and Lech Madeyski. "Empirical evaluation of multiobjective optimization algorithms searching for higher order mutants." Cybernetics and Systems 47, no. 1-2 (2016): 48-68.

[2] Papadakis, Mike, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults." In Proceedings of the 40th International Conference on Software Engineering, pp. 537-548. 2018.

[3] Papadakis, Mike, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. "Mutation testing advances: an analysis and survey." In Advances in Computers, vol. 112, pp. 275-378. Elsevier, 2019.

[4] Omar, Elmahdi, Sudipto Ghosh, and Darrell Whitley. "Comparing search techniques for finding subtle higher order mutants." In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pp. 1271-1278. 2014.

[5] Nayak, Janmenjoy, Bighnaraj Naik, and H. S. Behera. "A novel chemical reaction optimization based higher order neural network (CRO-HONN) for nonlinear classification." Ain Shams Engineering Journal 6, no. 3 (2015): 1069-1091.

[6] Lam, Albert YS, and Victor OK Li. "Chemical reaction optimization: a tutorial." Memetic Computing 4 (2012): 3-17.

[7] Lam, Albert YS, Jialing Xu, and Victor OK Li. "Chemical reaction optimization for population transition in peer-to-peer live streaming." In IEEE Congress on Evolutionary Computation, pp. 1-8. IEEE, 2010.

[8] James, J. Q., Albert YS Lam, and Victor OK Li. "Real-coded chemical reaction optimization with different perturbation functions." In 2012 IEEE Congress on Evolutionary Computation, pp. 1-8. IEEE, 2012.

[9] Abuljadayel, Anas, and Fadi Wedyan. "An approach for the generation of higher order mutants using genetic algorithms." Int. J. Intell. Syst. Appl.(IJISA) 10, no. 1 (2018): 34-35.

[10] Papadakis, Mike, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. "Mutation testing advances: an analysis and survey." In Advances in Computers, vol. 112, pp. 275-378. Elsevier, 2019.

[11] Dang, Xiangying, Dunwei Gong, Xiangjuan Yao, Tian Tian, and Huai Liu. "Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm." IEEE Transactions on Software Engineering 48, no. 6 (2021): 2141-2156.

[12] Jatana, Nishtha, and Bharti Suri. "An improved crow search algorithm for test data generation using search-based mutation testing." Neural Processing Letters 52 (2020): 767-784.

[13] Oh, Saeyoon, Seongmin Lee, and Shin Yoo. "Effectively sampling higher order mutants using causal effect." In 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 19-24. IEEE, 2021.

[14] Diniz, João P., Chu-Pan Wong, Christian Kästner, and Eduardo Figueiredo. "Dissecting Strongly Subsuming Second-Order Mutants." In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 171-181. IEEE, 2021.

[15] Amare, Melashu, Sudhir Kumar Mohapatra, and Tarini Prasad Panigrahy. "A Systematic Literature Review on Generating Higher-Order Mutant." In 2021 8th International Conference on Computing for Sustainable Global Development (INDIACom), pp. 131-136. IEEE, 2021.

[16] Nguyen, Quang-Vu, and Hai-Bang Truong. "An improvement of applying multi-objective optimization algorithm into higher order mutation testing." In Advanced Computational Methods for Knowledge Engineering: Proceedings of the 6th International Conference on Computer Science,

Applied Mathematics and Applications, ICCSAMA 2019 6, pp. 361-369. Springer International Publishing, 2020.

[17] Nguyen, Quang-Vu, and Hai-Bang Truong. "An improvement of applying multi-objective optimization algorithm into higher order mutation testing." In Advanced Computational Methods for Knowledge Engineering: Proceedings of the 6th International Conference on Computer Science, Applied Mathematics and Applications, ICCSAMA 2019 6, pp. 361-369. Springer International Publishing, 2020.

[18] Ghiduk, Ahmed S., and M. Rokaya. "An empirical evaluation of the subtlety of the data-flow based higher-order mutants." J. Theor. Appl. Inf. Technol. 97, no. 15 (2019): 4061-4074.

[19] do Prado Lima, Jackson Antonio, and Silvia Regina Vergilio. "A systematic mapping study on higher order mutation testing." Journal of Systems and Software 154 (2019): 92-109.

[20] Ghiduk, Ahmed S., and S. F. El-Zoghdy. "CHOMK: Concurrent higher-order mutants killing using genetic algorithm." Arabian Journal for Science and Engineering 43 (2018): 7907-7922.

[21] Ghiduk, Ahmed S., Moheb R. Girgis, and Marwa H. Shehata. "Reducing the cost of higher-order mutation testing." Arabian Journal for Science and Engineering 43 (2018): 7473-7486.

[22] Do, Van-Nho, Quang-Vu Nguyen, and Thanh-Binh Nguyen. "Evaluating mutation operator and test case effectiveness by means of mutation testing." In Intelligent Information and Database Systems: 13th Asian Conference, ACIIDS 2021, Phuket, Thailand, April 7–10, 2021, Proceedings 13, pp. 837-850. Springer International Publishing, 2021.

[23] Rahman, Mizanur, Kamal Z. Zamli, Md. Abdul Kader, Roslina Mohd Sidek, and Fakhrud Din. 2023. "Comprehensive Review on the State-of- the-Arts and Solutions to the Test Redundancy Reduction Problem With Taxonomy". Journal of Advanced Research in Applied Sciences and Engineering Technology 35 (1):62-87. https://doi.org/10.37934/araset.34.3.6287.

[24] Tengku Sulaiman, Tengku Mohd Sharir, Mohamad Minhat, Saiful Bahri Mohamed, Ahmad Syafiq Mohamed, Ahmad Ridhuan Mohamed, and Siti Nurul Akmal Yusof. 2020. "File and PC-Based CNC Controller Using Integrated Interface System (I2S)". Journal of Advanced Research in Applied Mechanics 70 (1):1-8. https://doi.org/10.37934/aram.70.1.18.

[25] Atamamen, Fidelis Osagie, Abdul Hakim Mohammed, and Temitope Folasade Atamamen. 2018. "Testing Measurement Invariance for Green Cleaning Services Implementation across Malaysian Cleaning Industry Stakeholders' Group". Progress in Energy and Environment 5 (May):50-61. https://www.akademiabaru.com/submit/index.php/progee/article/view/1045.

[26] Iman Fitri Ismail, Akmal Nizam Mohammed, Bambang Basuno, Siti Aisyah Alimuddin, and Mustafa Alas. 2022. "Evaluation of CFD Computing Performance on Multi-Core Processors for Flow Simulations". Journal of Advanced Research in Applied Sciences and Engineering Technology 28 (1):67-80.https://doi.org/10.37934/araset.28.1.6780.

[27] Habtemariam, Getachew Mekuria, and Sudhir Kumar Mohapatra. "A genetic algorithm-based approach for test case prioritization." In Information and Communication Technology for Development for Africa: Second International Conference, ICT4DA 2019, Bahir Dar, Ethiopia, May 28-30, 2019, Revised Selected Papers 2, pp. 24-37. Springer International Publishing, 2019.

[28] Getachew, Daniel, Sudhir Kumar Mohapatra, and Subhasish Mohanty. "A Heuristic-Based Test Case Prioritization Algorithm Using Static Metrics." In Optimization of Automated Software Testing Using Meta-Heuristic Techniques, pp. 45-58. Cham: Springer International Publishing, 2022.

[29] Mohapatra, Sudhir Kumar, and Srinivas Prasad. "A Chemical Reaction Optimization Approach to Prioritize the Regression Test Cases of Object-Oriented Programs." Journal of ICT Research & Applications 11, no. 2 (2017).

[30] Mohapatra, Sudhir Kumar, Arnab Kumar Mishra, and Srinivas Prasad. "Intelligent Local Search for Test Case Minimization." Journal of The Institution of Engineers (India): Series B 101, no. 5

(2020): 585-595.