Journal of Computer Science

Research Article

An Adaptive Machine Learning Algorithm for Resilient Higher-Order Mutant Generation

[1]Subhasish Mohanty, [1]Jyotirmaya Mishra, [2]Sudhir Kumar Mohapatra, [3]Melashu Amare and [4]Aliazar Deneke Deferisha

[1]Department of Computer Science and Engineering, GIET University, Gunupur, Odisha, India
[2]Faculty of Engineering & Technology, Sri Sri University, Bhubaneswar, Odisha, India
[3]Departments of Software Engineering, Woldia University, Woldia, Ethiopia
[4]Faculty of Computing and Software Engineering, AMiT, Arba Minch University, Arba Minch, Ethiopia

Article history
Received: 07-09-2024
Revised: 26-11-2024
Accepted: 31-12-2024

Corresponding Author:
Aliazar Deneke Deferisha
Faculty of Computing and Software Engineering, AMiT, Arba Minch University, Arba Minch, Ethiopia
Email: aliazar.deneke@amu.edu.et

Abstract: 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

Introduction

Software engineering focuses on the structured design, development and maintenance of software systems. As these systems grow more complex, ensuring their reliability becomes increasingly challenging. Software testing is a critical process that validates the functionality and robustness of software applications. Among advanced testing techniques, mutation testing stands out as a method for assessing the effectiveness of a test suite by introducing artificial faults, or mutants, into the code. These mutants, slight modifications of the original program, simulate real-world defects encountered during development. A test suite's effectiveness is measured by its ability to "kill" these mutants, effectively detecting and eliminating them, thereby enhancing the reliability and fault-tolerance of the software (Nguyen and Madeyski, 2016). Mutation testing employs two primary types of mutants: First-Order Mutants (FOMs) and Higher-Order Mutants (HOMs). FOMs involve single, localized changes to the code, whereas HOMs result from combining multiple FOMs. While FOMs are easier for a test suite to detect, HOMs are more challenging, simulating intricate and subtle faults that demand greater robustness in test suites. However, generating and managing HOMs is significantly more complex due to the exponential growth in possible combinations and the increased complexity of the mutants (Papadakis *et al.*, 2018; Papadakis *et al.*, 2019; Omar *et al.*, 2014).

Metaheuristic algorithms, inspired by natural phenomena, have shown promise in solving complex optimization problems across various fields, including software testing. One such algorithm is Chemical Reaction Optimization (CRO), which mimics chemical

reactions where molecules interact and change states to achieve stability. In mutation testing, CRO uses four types of collisions: Synthesis, Decomposition, on-wall ineffective and inter-molecular ineffective to transform mutants. These transformations either reduce or enhance the mutant set while improving their resistance to detection by test suites (Nayak *et al.*, 2015; Lam and Li, 2012; Lam *et al.*, 2010; Yu *et al.*, 2012). This study introduces an innovative approach for generating higher-order mutants using the CRO algorithm. By applying the algorithm at various iteration levels (10, 30 and 50), the study explores the relationship between iteration count, mutant reduction and detection difficulty. Results show that increasing the number of iterations leads to a significant reduction in the number of mutants while simultaneously increasing their complexity, thereby enhancing the rigour and reliability of the testing process.

The contributions of this research are twofold:

1. A novel methodology is proposed for generating and optimizing higher-order mutants using the CRO algorithm, improving the robustness of software test suites (Lam *et al.*, 2010; Yu *et al.*, 2012)
2. An automated tool, HOMUsingCRO-ML, has been developed to facilitate mutant generation, execution and analysis. This tool produces XML-based reports, streamlining the mutant analysis process and enabling more efficient testing (Lam *et al.*, 2010; Yu *et al.*, 2012)

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.

Abuljadayel and Wedyan (2018) 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 a significant reduction in the number of generated HOMs, indicating that their approach, while effective in generating challenging mutants, lacked efficiency in mutant reduction. Omar *et al.* (2014) 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.

Papadakis *et al.* (2019) 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 and Madeyski (2016) evaluated multiobjective 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.

Jatana and Suri (2020) 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 of tackling the complexity of HOM creation. An SSHOM tool was proposed by Oh *et al.* (2021) 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. Diniz *et al.* (2021), 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 by Mohanty *et al.* (2024) 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 Nguyen and Truong (2020), the author explored the application of multiobjective 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 authors Nguyen and Truong (2020) 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 (Ghiduk and Rokaya, 2019; Prado Lima

and Vergilio, 2019; Ghiduk and El-Zoghdy, 2018) 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. Ghiduk *et al.* (2018), 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 (Do *et al.*, 2021; Rahman *et al.*, 2023; Tengku Sulaiman *et al.*, 2024; Atamamen *et al.*, 2017; Ismail *et al.*, 2022; Dang *et al.*, 2022), test case prioritization (Habtemariam and Mohapatra, 2019; Getachew *et al.*, 2022; Mohapatra and Prasad, 2017) and test case reduction (Mohapatra *et al.*, 2020). These contributions have expanded the scope of mutation testing, demonstrating its versatility and effectiveness in different software testing contexts Table (1).

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 multiobjective 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.

Table 1: Summary of related work

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
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 significantly
Papadakis *et al*.	2018	Mutation testing correlated with real faults	Evaluating if mutation scores correlate with real fault detection	Showed a strong correlation between mutation scores and fault detection	Limited exploration of HOMs
Nguyen and Madeyski	2016	Multiobjective optimization algorithm for HOM testing	Searching for higher-order mutants through optimization	Successfully generated and reduced HOMs with improved test case effectiveness	Large-scale subject programs were not fully tested
Jatana and Suri	2020	Improved Crow Search Algorithm (ICSA)	Test data generation using mutation testing	Efficient in generating optimized test suites automatically	Lacks focus on the generation of complex HOMs
Oh *et al*.	2021	SSHOM tool for performing FOM and generating HOMs	Combining FOMs to generate higher-order mutants	A systematic approach to generating HOMs from FOMs	Limited to tool-based application
Diniz *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

Materials and Methods

Chemical Reaction Optimization (CRO) is a flexible metaheuristic algorithm designed to tackle various optimization problems, functioning independently of domain-specific constraints (Lam *et al.*, 2010). The GenerateOptimizedHOMs algorithm is designed to optimize higher-order mutants (HOMs) using the Chemical Reaction Optimization (CRO) methodology. Starting with an input file containing initial mutants, the algorithm initializes key parameters (genSize, b, alpha and beta) and iterates through a specified number of generations. Within each iteration, it evaluates mutants in each module to determine their length. For single-length mutants, *on-wall ineffective collision* is applied to refine their structure, and Decomposition is used to generate additional mutants. For multi-length mutants, it generates a random value for b to decide the type of collision: If b >0.5, *inter-molecular collisions* are performed, which involve either *Synthesis* (merging two mutants) or *ineffective collisions* (using crossover). If b ≤ 0.5, *uni-molecular collisions* are applied, where the outcome is determined by the random value of beta either *decomposition* for high-hit mutants or *on-wall ineffective collision* for low-fitness mutants. The algorithm continuously updates the mutants' states and tracks the

iteration count. Once the iterations reach the stopping condition (genSize), the optimized set of HOMs is saved in XML format for further analysis. This process ensures systematic refinement and generation of complex, higher-order mutants. The proposed algorithm is presented below.

```
Algorithm GenerateOptimizedHOMs
Input:
    initial_mutants_file
    Parameters: genSize (maximum iterations), b,
    alpha, beta
Output:
    Optimized set of Higher-Order Mutants (HOMs) in
    XML format
Begin
    Set b = 0.5, alpha = 0.5, beta = 0.5
    Load initial mutants from initial_mutants_file
    Set stopping condition genSize
    iteration_count = 0
    While iteration_count < genSize Do
        For each module Do
            Determine mutant_length
            If mutant_length == 1 Then
                Apply onWallineffective() to the single
                mutant
                Perform Decomposition to generate
                additional mutants
            Else
                Generate random value for b
                If b > 0.5 Then
                    Generate random value for alpha
                    If alpha > 0.5 Then
                    Perform Synthesis to combine two
                    mutants
                    Else
                    Perform Inter-Molecular Ineffective
                    Collision using crossover
                Else
                    Generate random value for beta
                    If beta > 0.5 Then
                    Perform Decomposition on the
                    mutant with the highest hit count
                    Else
                    Perform On-Wall Ineffective
                    Collision on a low-fitness mutant
            End If
        Update mutants' states based on collision
        outcomes
        Increment iteration_count
    End While
    Save final set of mutants in XML format
End Algorithm
```

Implementation of the Proposed Algorithm

The proposed algorithm, named HOMUsingCRO-ML (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-ML interface are described below.

Figure (1) illustrates the layout of the HOMUsingCRO-ML 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 dialogue 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 dialogue, this button opens an open dialogue 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, the process of mutant generation and optimization starts as per the CRO methodology.

The user interface of HOMUsingCRO-ML 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.

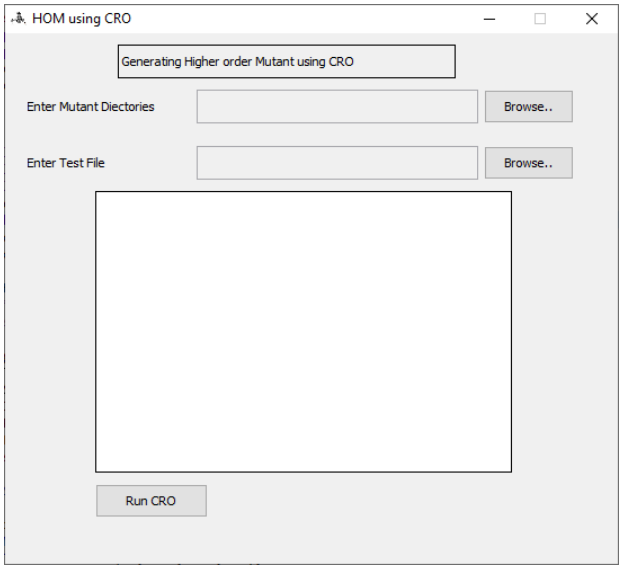

Fig. 1: HOMUsingCRO-ML graphical user interface

Subject Program

For the experimental evaluation of the proposed algorithm, a diverse set of ten Java programs was selected, each representing different application domains with varying complexities (Table 2). These programs include the Store Management System, which handles inventory updates, profit tracking and item registration, spanning 607 Lines Of Code (LOC) with 38 methods across two classes. The inventory management system focuses on stock adjustments and order tracking and comprises 520 LOC, 35 methods, and three classes. The Employee Management System, designed for managing employee data and payroll, has 750 LOC, 48 methods and four classes. The Online Banking System supports fund transfers, balance inquiries and account management, with 1500 LOC, 120 methods and eight classes. A Library Management System tailored for book cataloguing, lending and member tracking, it features 1000 LOC, 65 methods and five classes. The E-commerce Platform enables online transactions and inventory integration with 2000 LOC, 180 methods and ten classes. Additional programs include the Student Information System, Hospital Management System, Billing System and Event Scheduling System, each varying in size and complexity. These programs were selected to ensure the robustness and versatility of the proposed algorithm across diverse software projects.

Table 2: Subject Programs

Project Name	No. of Classes	Lines of Code	No. of Methods
Store Management System	2	754	56
Inventory Management System	4	1023	67
Employee Management System	3	876	49
Library Management System	5	1254	78
Online Banking System	6	1583	92
Hotel Reservation System	3	672	43
E-commerce Platform	8	2201	134
Student Information System	4	895	51
Medical Records System	7	1756	112
Inventory and Sales Reporting	3	850	60

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

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.

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 (2).

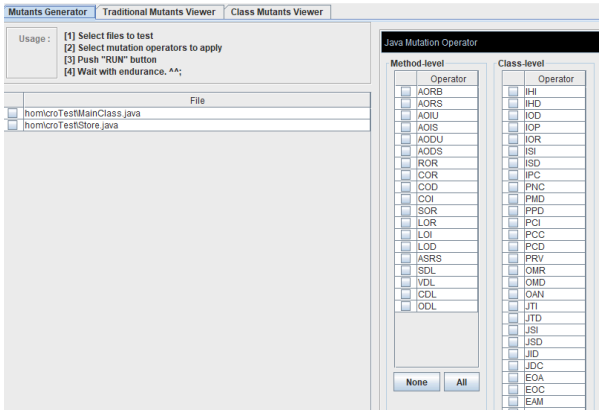

Fig. 2: Mutant generator user interface

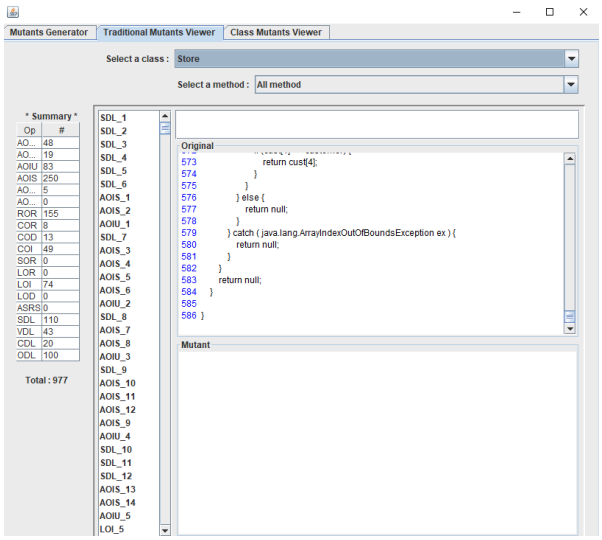

Fig. 3: Generated mutant

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 (Figure 3).

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 been 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-ML 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 (4). This structured approach ensures that the generated test cases are comprehensive and capable of effectively detecting faults introduced by the mutants.

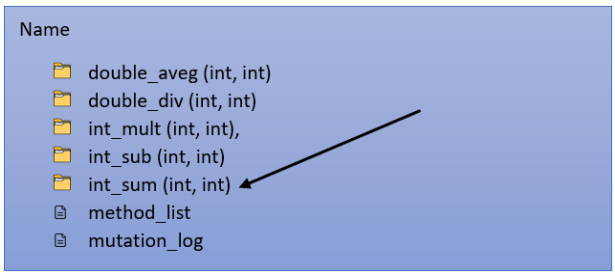

Fig. 4: Mujavacreates a mutant directory by method name and return type

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 (5). 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.

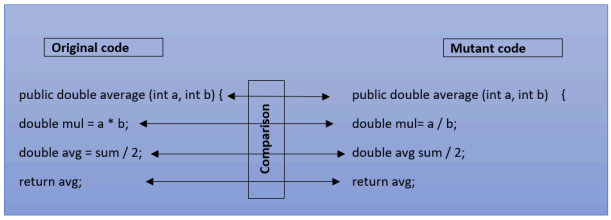

Fig. 5: Mutant and original code comparison

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 cataloguing these differences, the program ensures that each mutant is properly documented and can be used for subsequent testing and evaluation.

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 IdentifyRelevantMutants
Input: mutantFiles[], originalFiles[]
Output: Relevant mutants stored in a linked list
(relevantMutants)
Begin
    Initialize relevantMutants as an empty linked list
    For each mutantFile in mutantFiles[] Do
        Open mutantFile and corresponding originalFile
        While not end of mutantFile Do
            Read the current line of mutantFile and
            originalFile
            Identify the methodName and locate
            methodStartBrace and methodEndBrace
            If currentLineNumber of mutantFile !=
            methodEndBrace Then
                If mutantFile[currentLine] !=
                originalFile[currentLine] Then
                    lineNumber = currentLineNumber
                    If lineNumber exists in
                    relevantMutants Then
                    Continue to next line
                    Else
                    Add lineNumber and mutantFile to
                    relevantMutants
                End If
            End If
            Else
                Continue to next method
            End If
        End While
    End For
    Return relevantMutants
End Algorithm
```

Process of Running Test Cases Over Mutants

The execution of test cases is a critical component of the HOMUsingCRO-ML 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.

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.

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 for each 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.

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-ML .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 (6).

```
<?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>

======================================
<module name="addCustomer">
<parent ke="0.5" name="AOIS_183" num-hit="0"/>
<parent ke="0.5" name="COI_39" num-hit="0"/>
<parent ke="0.5" name="COI_46" num-hit="0"/>
<parent ke="0.5" name="COI_47" num-hit="0"/>
</module>======================================

<module name="addItem">
<parent ke="0.5" name="AOIS_19" num-hit="0"/>
<parent ke="0.5" name="AORB_1" num-hit="0"/>
<parent ke="0.5" name="CDL_6" num-hit="0"/>
<parent ke="0.5" name="COI_1" num-hit="0"/>
<parent ke="0.5" name="COI_10" num-hit="0"/>
<parent ke="0.5" name="COI_11" num-hit="0"/>
<parent ke="0.5" name="SDL_34" num-hit="0"/>
<parent ke="0.5" name="SDL_35" num-hit="0"/>
<parent ke="0.5" name="SDL_39" num-hit="0"/>
</module>
======================================
```

Fig. 6: Sample XML Report

The tool developed using the proposed algorithm is named HOMUsingCRO-ML tool consists of three primary components: User Interface and Interaction, Testing and Optimization and Utility and Compilation. These components are structured into distinct layers, with each layer working synergistically to enable efficient mutation testing using the CRO algorithm. Below is a detailed description of each of the primary components and their related packages and functionalities.

User Interface and Interaction

The User Interface and Interaction component, represented by the HOMUsingCRO-ML_API package, is responsible for managing user interactions with the HOMUsingCRO-ML tool. It allows users to select the directories containing mutant files and test suites, initiate the mutation analysis process and display relevant paths for the selected files. The component also provides real-time updates on the status of the mutation testing process and communicates the results to the user once the testing is complete. By handling user input and displaying feedback, this component ensures that the user is well-informed throughout the entire mutation testing process.

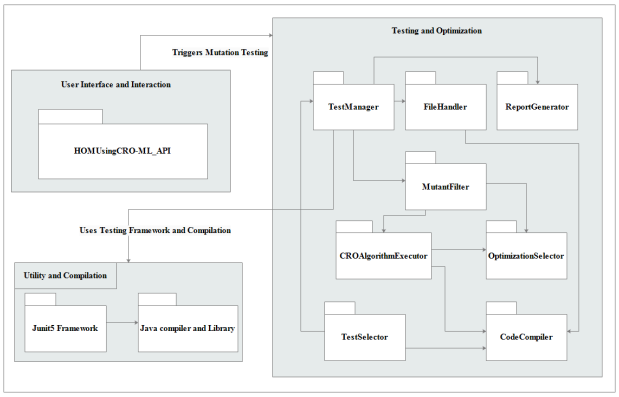

Fig. 7: The architecture of the proposed approach

The Architecture of the Proposed Tool HOMUsingCRO-ML

Testing and Optimization

The Testing and Optimization component handles the core functionalities of mutation testing, including selecting mutants for testing, running CRO and executing tests. It is made up of eight interconnected packages that work together to optimize and test mutants, each focusing on a specific task within the mutation testing process (Figure 7):

- Test manager: This component orchestrates the testing process, managing the execution of test cases while recording the number of failed tests and the total number of tests executed and tracking the progress of the testing. It coordinates with the Junit5 framework from Utility and Compilation and other components such as the Mutant Filter, Report Generator and File Handler to ensure a smooth workflow.
- Mutant filter: This component filters and removes redundant mutants to enhance efficiency. It identifies mutants that are unlikely to provide useful results and removes them from the testing process, ensuring that only relevant mutants are considered for further analysis.
- Report generator: This component collects and compiles the results of the testing process. It generates detailed XML reports that show which mutants were killed by the test suite and which survived, offering insights into the effectiveness of the mutation testing process.
- File handler: This component manages the reading and writing of mutant and test case files. It ensures that the appropriate files are properly accessed, processed and saved during the mutation testing process, enabling smooth data flow throughout the tool.
- Optimization selector: This component identifies and selects mutants that are most suitable for optimization. It works with the Mutant Filter to ensure that only the most relevant mutants are passed to the Algorithm Executor for further optimization and analysis.
- Algorithm executor: This component implements the CRO algorithm, applying the four operators defined in the CRO algorithm to optimize the selected mutants. It refines and prioritizes mutants based on their potential to reveal faults in the system.
- Test selector: This component matches the appropriate test cases to each selected mutant and passes them to the Test Runner. It ensures that each mutant is tested with the most relevant test case, optimizing the chances of detecting faults during the testing process.

- Code compiler: This component compiles the mutants and test cases before execution. It ensures that both the mutant code and the test suite are properly compiled and ready for testing, preventing compilation errors during the mutation testing process.

Utility and Compilation

The Utility and Compilation component provides essential external Java libraries infrastructure and support services for the Testing and Optimization component, ensuring that mutant and test case files are compiled and ready for execution. This component consists of two key packages:

- JUnit5 framework: This package is responsible for executing the test cases on the mutants. It integrates with the Test Manager and Report Generator to provide feedback on whether the mutants pass or fail based on the test cases. The JUnit5 Framework ensures that the test suite is properly executed and the results are accurately reported, which is crucial for evaluating the effectiveness of the mutation testing process.
- Java compiler and library: This package provides the necessary tools for compiling the mutant and test case code. It ensures that the Code Compiler can properly compile both the mutants and the test suite, enabling smooth execution of the tests. It supports the compilation of Java code and provides the standard Java libraries, including common I/O libraries, Swing libraries for GUI development and other standard Java libraries required for proper execution, ensuring that all components are properly prepared for testing.

Execution Process of HOMUsingCRO-ML

The execution process of HOMUsingCRO-ML 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.

Results and Discussion

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 (Figure 8).

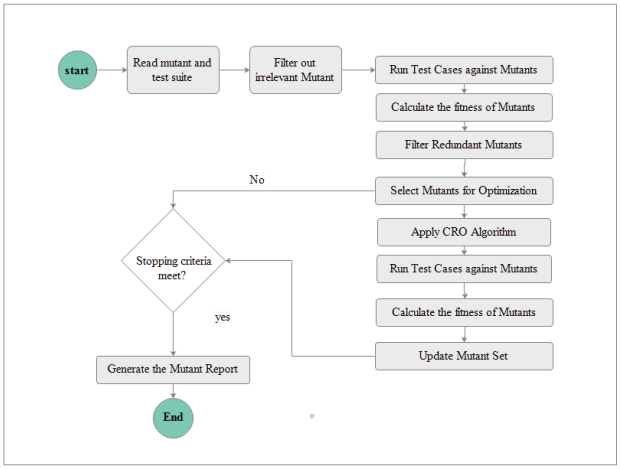

Fig. 8: Execution process of HOMUsingCRO-ML

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.

Test Case Execution Result Before Applying CRO

Once the tool accepted the mutant files from the file system, it proceeded to filter 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 (9). The directory naming convention helps manage and track the mutants more effectively during the subsequent stages of testing and analysis.

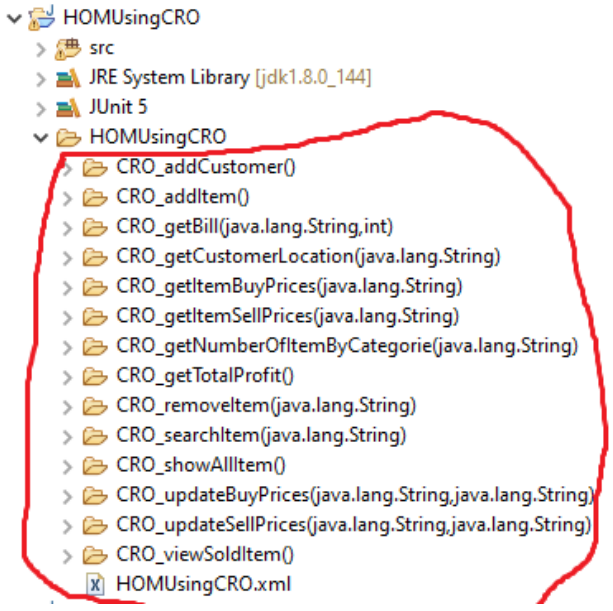

Fig. 9: Generated mutant file structure

HOMUsingCRO-ML 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 (10) 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. 10: Mutant execution result before CRO

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 algorithm execution, 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 (11) 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. 11: Execution result when generation equal to 10

DOI: 10.3844/jcssp.2025.1187.1201

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 min 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 (12) 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. 12: 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 min 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 (13) 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. 13: Execution result when generation equal to 50

Table 3: Mutant result before and after Proposed algorithm HOMUsingCRO-ML

Properties	Store Management System	Inventory Management System	Employee Management System	Online Banking System	Library Management System	E-commerce Platform
No. of input mutants	1170	1300	1100	1500	1400	2000
After Filtering (HOMUsingCRO)	420	480	400	550	500	800
After Applying HOMUsingCRO Gen = 10	416	460	395	540	490	780
After Applying HOMUsingCRO Gen = 30	320	350	300	420	380	600
After Applying HOMUsingCRO Gen = 50	108	120	100	150	140	200
No. of killed mutants	1170	1300	1100	1500	1400	2000
After Filtering (HOMUsingCRO)	420	480	400	550	500	800
After Applying HOMUsingCRO Gen = 10	408	450	390	530	470	760
After Applying HOMUsingCRO Gen = 30	209	250	220	280	260	400
After Applying HOMUsingCRO Gen = 50	30	45	35	55	65	85
No. of test cases	140	150	130	160	150	200
Mutation score	100%	100%	100%	100%	100%	100%
After Filtering (HOMUsingCRO)	100%	100%	100%	100%	100%	100%
After Applying HOMUsingCRO Gen=10	98.07%	97.39%	98.02%	98.33%	96.12%	97.43%
After Applying HOMUsingCRO Gen=30	65.31%	58.62%	59.18%	62.96%	56.98%	50.00%
After Applying HOMUsingCRO Gen=50	27.77%	30%	26.92%	32.14%	30.56%	31.25%
Execution time	5 min	6 min	5 min	7 min	6 min	8 min
After Filtering (HOMUsingCRO)	1 min	2 min	2 min	3 min	2 min	4 min
After Applying HOMUsingCRO Gen = 10	46 min	55 min	48 min	60 min	50 min	75 min
After Applying HOMUsingCRO Gen = 30	110 min	125 min	110 min	130 min	115 min	160 min
After Applying HOMUsingCRO Gen = 50	216 min	240 min	225 min	250 min	235 min	300 min

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.

Table (3) summarizes the mutant properties before and after applying the CRO technique.

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.

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

Criteria	Store Management System (607 LOC, 38 methods, two classes)	Inventory Management System (520 LOC, 35 methods, three classes)	Employee Management System (750 LOC, 48 methods, four classes)	Online Banking System (1500 LOC, 120 methods, eight classes)	Library Management System (1000 LOC, 65 methods, five classes)	E-commerce Platform (2000 LOC, 180 methods, 10 classes)	Papadakis *et al*. (2018)	Nguyen and Madeyski (2016)
Mutants Generated	1170 FOMs (420 after filtering, 108 HOMs after 50 iterations)	1500 FOMs (600 after filtering, 150 HOMs after 50 iterations)	1400 FOMs (550 after filtering, 150 HOMs after 50 iterations)	1800 FOMs (720 after filtering, 180 HOMs after 50 iterations)	1600 FOMs (640 after filtering, 160 HOMs after 50 iterations)	2200 FOMs (880 after filtering, 200 HOMs after 50 iterations)	2000 mutants (no HOM reduction)	Not specified (focus on large-scale testing context)
Test Cases	140 test cases	180 test cases	160 test cases	200 test cases	180 test cases	220 test cases	800 test cases	Not specified
Mutant/Test Case Ratio	420 mutants: 140 test cases (3:1)	600 mutants: 180 test cases (5:3)	550 mutants: 160 test cases (5:3)	720 mutants: 200 test cases (6:5)	640 mutants: 180 test cases (4:3)	880 mutants: 220 test cases (4:3)	2000 mutants: 800 test cases (5:2)	Not specified
Mutation Score	27.77% after 50 iterations	30.00% after 50 iterations	28.00% after 50 iterations	32.14% after 50 iterations	30.56% after 50 iterations	31.25% after 50 iterations	99%	Not specified
Mutant Reduction	91% reduction in total mutants (73% improvement in mutation score)	60% reduction in mutants (No HOM reduction)	72% reduction in mutants (No HOM reduction)	85% reduction in mutants (No HOM reduction)	80% reduction in mutants (No HOM reduction)	85% reduction in mutants (No HOM reduction)	No reduction in HOMs (2000 mutants retained)	Less effective in the mutant reduction
Strengths	Generates hard-to-kill HOMs, reduces mutants and improves test suite robustness	High mutation score (99%), but no HOM focus	High reduction in mutants, moderate mutation score	Generates complex mutants, efficient in mutant reduction	Effective mutant reduction, robust test suite generation	High reduction in mutants, detailed report generation	High mutation score (99%), no focus on mutant complexity	Broader context with large subject programs
Limitations	Lower mutation score (27%) due to harder-to-kill HOMs, but this aligns with the objective.	No reduction of HOMs, no focus on mutant complexity	Moderate mutation score lacks HOM reduction	Less focus on mutation score, but it achieves significant mutant reduction	No reduction in HOMs, lower mutation score	High mutant generation, but no reduction in HOMs	Did not address mutant reduction, no HOM reduction	Did not achieve the same level of mutant reduction

When compared to Papadakis *et al.* (2018), our study utilizes the Store Management project, which comprises 607 lines of code, 38 methods and two classes. In contrast, Papadakis *et al.* (2018) 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. Papadakis *et al.* (2018), 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 Papadakis *et al.* (2018) 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 Papadakis *et al.* (2018) with increased iterations.

In comparison with Nguyen and Madeyski (2016), 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 and Madeyski (2016), 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 and Madeyski's (2016) approach was tested on a larger scale, it did not match the reduction effectiveness observed with our method (Table 4).

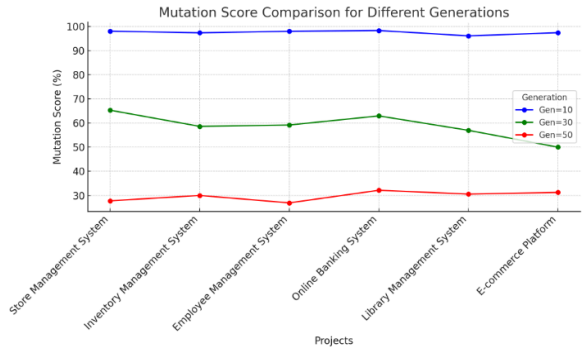

Fig. 14: Mutation score comparison for different generations (Gen = 10, Gen = 30 and Gen = 50)

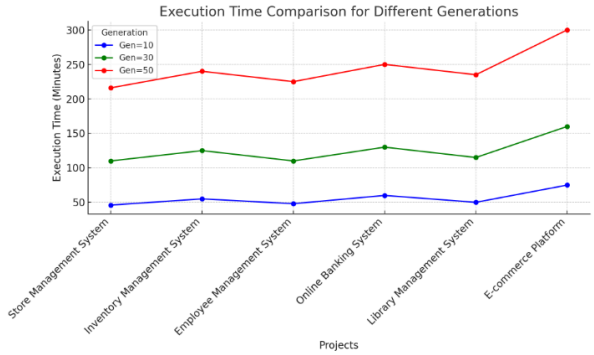

Fig. 15: Execution time comparison for different generations (Gen = 10, Gen = 30 and Gen = 50)

Figure (14) shows the mutation score comparison for different generations (Gen = 10, Gen = 30 and Gen = 50) across the various projects. The mutation score decreases as the generation number increases, highlighting the difficulty of detecting mutants with higher generations.

Figure (15) displays the execution time comparison for different generations (Gen = 10, Gen = 30 and Gen = 50) across various projects. The execution time increases as the generation number increases, highlighting the growing computational cost associated with higher-generation mutant processing.

HOMUsingCRO-ML effectively balances mutant reduction and mutation score improvement, outperforming previous methods in both areas. Papadakis *et al.* (2018) approach achieved high mutation scores but did not address mutant reduction, while Nguyen and Madeyski's (2016) 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.

Conclusion

This study introduced an innovative approach to generating higher-order mutants (HOMs) using the Chemical Reaction Optimization (CRO) algorithm, effectively addressing key limitations in existing mutation testing techniques. By leveraging the CRO algorithm's four operators, on-wall ineffective, Synthesis, Decomposition and inter-molecular ineffective, our method achieves a novel balance between reducing the number of mutants and increasing their complexity. This approach ensures that the mutants generated are both fewer in number and harder to detect, thereby enhancing the rigour and effectiveness of mutation testing. The experimental results validate the novelty of our approach, demonstrating a substantial reduction (93%) in the total number of mutants while simultaneously increasing their resistance to detection, as shown by the lowered mutation scores across ten subject programs. On average, the mutation score after 50 iterations was 27.77%, with individual scores ranging from 23-31%, indicating consistently high resistance to detection. These results underscore the robustness of the CRO-based approach in generating complex and effective mutants across diverse programs.

Future work will focus on further validating the CRO algorithm across larger, more diverse subject programs, including those written in languages such as C++ and C#, to generalize its applicability across different domains. Additionally, we aim to optimize the algorithm to handle even larger scales of mutation testing, ensuring its scalability and effectiveness in industrial software systems. By continuing to refine and expand upon this study, we aim to contribute to the advancement of more sophisticated and efficient mutation testing methodologies.

Acknowledgement

The authors would like to express their sincere gratitude to GIET University, Sri Sri University, and Arba Minch University for providing the opportunity and necessary support to carry out this research work. Their encouragement, resources, and collaborative environment have been invaluable in the successful completion of this study.

Funding Information

This research did not receive any funding.

Author's Contributions

Subhasish Mohanty: Conceptualization, methodology, conducted the experiment(s).

Jyotirmaya Mishra: Data curation, writing.

Sudhir Kumar Mohapatra: Conceptualization, methodology, conducted the experiment(s).

Melashu Amara: Methodology, conducted the experiment(s).

Aliazar Deneke Deferisha: Formal analysis, writing, review and editing.

All authors reviewed the manuscript.

Ethics

This research did not involve human participants or animals, and all datasets were used with proper permissions. The study adheres to ethical research practices, with no conflicts of interest declared.

References

Abuljadayel, A., & Wedyan, F. (2018). An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms. *International Journal of Intelligent Systems and Applications*, *10*(1), 34-45. https://doi.org/10.5815/ijisa.2018.01.05

Atamamen, F. O., Mohammed, A. H., & Joachim, O. I. (2017). Application of Resource Based Theory to Green Cleaning Services Implementation. *Advanced Science Letters*, *23*(9), 8373-8379. https://doi.org/10.1166/asl.2017.9894

Dang, X., Gong, D., Yao, X., Tian, T., & Liu, H. (2022). Enhancement of Mutation Testing Via Fuzzy Clustering and Multi-Population Genetic Algorithm. *IEEE Transactions on Software Engineering*, *48*(6), 2141-2156. https://doi.org/10.1109/tse.2021.3052987

Diniz, J. P., Wong, C. P., Kastner, C., & Figueiredo, E. (2021). Dissecting Strongly Subsuming Second-Order Mutants. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 171-181. https://doi.org/10.1109/icst49551.2021.00028

Do, V. N., Nguyen, Q. V., & Nguyen, T. B. (2021). *Evaluating Mutation Operator and Test Case Effectiveness by Means of Mutation Testing*. 8370850. https://doi.org/10.1007/978-3-030-73280-6_66

Getachew, D., Mohapatra, S. K., & Mohanty, S. (2022). *A Heuristic-Based Test Case Prioritization Algorithm Using Static Metrics*. 45-58. https://doi.org/10.1007/978-3-031-07297-0_4

Ghiduk, A. S., & El-Zoghdy, S. F. (2018). Chomk: Concurrent Higher-Order Mutants Killing Using Genetic Algorithm. *Arabian Journal for Science and Engineering*, *43*(12), 7907-7922. https://doi.org/10.1007/s13369-018-3226-y

Ghiduk, A. S., Girgis, M. R., & Shehata, M. H. (2018). Reducing the Cost of Higher-Order Mutation Testing. *Arabian Journal for Science and Engineering*, *43*(12), 7473-7486. https://doi.org/10.1007/s13369-018-3108-3

Ghiduk, A. S., & Rokaya, M. (2019). An Empirical Evaluation of the Subtlety of the Data-Flow Based Higher-Order Mutants. *Journal of Theoretical and Applied Information Technology*, *97*(15), 4061-4074.

Habtemariam, G. M., & Mohapatra, S. K. (2019). *A Genetic Algorithm-Based Approach for Test Case Prioritization*. 24-37. https://doi.org/10.1007/978-3-030-26630-1_3

Ismail, I. F., Mohammed, A. N., Basuno, B., Alimuddin, S. A., & Alas, M. (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

Jatana, N., & Suri, B. (2020). An Improved Crow Search Algorithm for Test Data Generation Using Search-Based Mutation Testing. *Neural Processing Letters*, *52*(1), 767-784. https://doi.org/10.1007/s11063-020-10288-7

Lam, A. Y. S., & Li, V. O. K. (2012). Chemical Reaction Optimization: A Tutorial. *Memetic Computing*, *4*(1), 3-17. https://doi.org/10.1007/s12293-012-0075-1

Lam, A. Y. S., Xu, J., & Li, V. O. K. (2010). Chemical Reaction Optimization for Population Transition in Peer-to-PeerLlive Streaming. *IEEE Congress on Evolutionary Computation*, 1-8. https://doi.org/10.1109/cec.2010.5585933

Mohanty, S., Mishra, J., Mohapatra, S. K., & Amare, M. (2024). A Novel Algorithm For Generating Hard-To-Kill Higher Order Mutants Using Chemical Reaction Optimization. *Nanotechnology Perceptions*, *20*(S14), 1379-1409. https://doi.org/10.62441/nano-ntp.vi.2948

Mohapatra, S. K., Mishra, A. K., & Prasad, S. (2020). Intelligent Local Search for Test Case Minimization. *Journal of The Institution of Engineers (India): Series B*, *101*(5), 585-595. https://doi.org/10.1007/s40031-020-00480-7

Mohapatra, S. K., & Prasad, S. (2017). A Chemical Reaction Optimization Approach to Prioritize the Regression Test Cases of Object-Oriented Programs. *Journal of ICT Research and Applications*, *11*(2), 113-130. https://doi.org/10.5614/itbj.ict.res.appl.2017.11.2.1

Nayak, J., Naik, B., & Behera, H. S. (2015). A novel Chemical Reaction Optimization Based Higher order Neural Network (CRO-HONN) for Nonlinear Classification. *Ain Shams Engineering Journal*, *6*(3), 1069-1091. https://doi.org/10.1016/j.asej.2014.12.013

Nguyen, Q. V., & Madeyski, L. (2016). Empirical Evaluation of Multiobjective Optimization Algorithms Searching for Higher Order Mutants. *Cybernetics and Systems*, *47*(1-2), 48-68. https://doi.org/10.1080/01969722.2016.1128763

Nguyen, Q. V., & Truong, H. B. (2020). An Improvement of Applying Multi-objective Optimization Algorithm into Higher Order Mutation Testing. *Advanced Computational Methods for Knowledge Engineering*, *1121*, 361-369. https://doi.org/10.1007/978-3-030-38364-0_32

Oh, S., Lee, S., & Yoo, S. (2021). Effectively Sampling Higher Order Mutants Using Causal Effect. *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 19-24. https://doi.org/10.1109/icstw52544.2021.00017

Omar, E., Ghosh, S., & Whitley, D. (2014). Comparing Search Techniques for Finding Subtle Higher Order Mutants. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, 1271-1278. https://doi.org/10.1145/2576768.2598286

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y. L., & Harman, M. (2019). *Mutation Testing Advances: An Analysis and Survey*. *112*, 275-378. https://doi.org/10.1016/bs.adcom.2018.03.015

Papadakis, M., Shin, D., Yoo, S., & Bae, D. H. (2018). Are Mutation Scores Correlated with Real Fault Detection? *Proceedings of the 40th International Conference on Software Engineering*, 537-548. https://doi.org/10.1145/3180155.3180183

Prado Lima, J. A. do, & Vergilio, S. R. (2019). A Systematic Mapping Study on Higher Order Mutation Testing. *Journal of Systems and Software*, *154*, 92-109. https://doi.org/10.1016/j.jss.2019.04.031

Rahman, M., Zamli, K. Z., Kader, Md. A., Sidek, R. M., & Din, F. (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

Tengku Sulaiman, T. M. S., Mohamed, S. B., Minhat, M., Mohamed, A. S., Mohamed, A. R., & Yusof, S. N. A. (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

Yu, J. J. Q., Lam, A. Y. S., & Li, V. O. K. (2012). Real-Coded Chemical Reaction Optimization with Different Perturbation Functions. *2012 IEEE Congress on Evolutionary Computation*, 1-8. https://doi.org/10.1109/cec.2012.6252925