UDC 004.9 **doi: 10.26906/SUNZ.2023.2.102-107**

Mykhailo Hulevych, Oleksii Kolomiitsev

National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine

AUTOMATED TEST GENERATION TECHNIQUES FOR C++ SOFTWARE

Abstract. Automation of test script generation is critically important in modern software quality assurance, particularly for complex languages such as C++, where manual testing becomes resource-intensive and error-prone. Automated approaches significantly reduce the testing effort, improve test effectiveness, and enhance overall software reliability. The CIDER tool, introduced by the author, offers a promising automated solution by generating test scenarios based on recorded program executions using harmony search for inputs optimization. Despite its benefits, tool faces next limitations: incomplete coverage of complex branching logic, inefficient exploration of large input spaces, and difficulties in handling semantically rich or contextually complex input data. The overview article aims to systematically explore, compare, and evaluate various automated test generation techniques such as symbolic execution, concolic testing, evolutionary algorithms, reinforcement learning, and model-based testing. The objectives are: to classify these methods based on specified criteria and identify suitable approaches that could enhance the tool's capability for automated test generation in terms of coverage efficiency.

Keywords: test automation, C++, regression testing, artificial bee colony, ant colony optimization, firefly algorithm, cuckoo search, q-learning.

Introduction

C++ is a foundational language for systems level and performance-critical software. It's widely used in domains such as embedded systems, operating systems, high-performance computing, financial, and real-time applications. However, the language's inherent complexity including manual memory management, template metaprogramming, and pointer arithmetic, introduces significant challenges for effective and reliable testing, particularly due to associated security risks. In response to the increasing demand for scalable and high-assurance software, the automation of test suite generation has become an essential focus in recent software engineering research. Ensurance the software quality requires additional attention to low-level constructs, tightly coupled modules, subtle language semantics, complex C++ memory model.

Thus, the study and advancement of automated test suite generation techniques is of paramount importance. The software systems are developed rapidly in Agile and DevOps environments and automated testing becomes a crucial enabler of continuous integration and regression safety. In the C++ software even minor changes can introduce subtle and difficult-to-detect bugs, automated test generation provides a reliable safety assurance for maintaining code quality and system reliability.

Recent publications. Research efforts continue to underline the importance of test automation, particularly for C++, that exhibits substantial structural and semantic complexity. Recently, Zhang et al. [1] introduced CITYWALK framework. The framework leverages large language models (LLMs) and language-specific aspects of knowledge to automate unit test generation, particularly for C++ codebases. It illustrates the growing role of artificial intelligence in addressing language-specific challenges. In addition, Rho et al. [2] presented the Coyote testing framework. It demonstrates the practical viability of concolic execution-based automation in industrial environments that achieves significant improvements in test coverage without manual intervention. Beyond individual tools and frameworks, broader empirical studies reinforce the trend. A quantitative analysis [3] showed that higher levels of test automation maturity are directly associated with improved software quality across industrial projects. These findings are echoed by recent industry reports, which highlight automation as a core element of scalable and sustainable quality assurance practices in contemporary software engineering. As project complexity and delivery speed increase, automated testing becomes not only a technical asset but also an imperative for ensuring robustness and quality [4].

At the same time, several critical challenges of automated test suite generation persist. These include the absence of reflective capabilities, the presence of non-determinism in standard libraries, and the lack of standardized mechanisms for specifying expected behavior across software components. These issues have motivated ongoing work in areas such as method call sequence mining, context-sensitive test generation, and mutation-based test evaluation [5]. Collectively, these research directions underscored both the feasibility and necessity of continued investment in automated test generation for C++, particularly as software complexity and safety requirements continue to grow.

Research objectives. The CIDER [6] tool is designed to address these challenges. It exemplifies a fully automated black-box testing approach tailored specifically for C++ libraries. It leverages recorded program executions to generate automated test scenarios, subsequently applying heuristic optimization techniques, such as harmony search, to improve test coverage. By providing compatibility with industry-standard libraries and facilitating flexible script-based testing approaches, the tool enables effective and realistic testing scenarios for complex software systems and demonstrates several advantages: effective initial scenario generation based on actual program execution, the use of harmony search to enhance test coverage, compatibility with industry-standard libraries and tools via bindings and scripts. However, the tool has incomplete coverage of complex branching logic due to limited systematic path exploration, motivating the investigation of additional enhancement techniques.

The research objectives are:

RO1: To explore and systematically compare various automated test generation techniques, including concolic and symbolic execution, evolutionary algorithms, reinforcement learning, model-based and search-based testing.

RO2: To identify methodologies and best practices that are most effective in overcoming CIDER's identified limitations.

In the next sections an overview of most popular automation techniques will be provided, along with their classification with respect to CIDER improvement relevance.

Classification Criteria

The criteria for classification include:

- complexity of control-flow
- complexity of data-flow
- semantical-rich input
- automation potential
- regression testing potential
- black-box testing suitability

For each criterion the description is provided in Table 1.

Table 1 – **Classification criteria description**

	Criteria	Description
Source code characteristics	Control-flow complexity	Measures the ability of a testing technique to effectively handle intricate branching logic and multiple execution paths within the source code.
	Data-flow complexity	Indicates how effectively a testing technique can track and analyze data dependencies, variable usage, and transformations throughout the software.
	Semantical rich input	Assesses the capability of the technique to generate meaningful, contextually valid input data, particularly for structured or domain-specific scenarios (e.g., JSON, XML, API payloads).
Tool operational characteristics	Automation potential	Evaluates the degree to which a testing technique can operate autonomously without manual intervention after the initial setup.
	Regression testing potential	Reflects the suitability of a technique to repeatedly validate software correctness over time, particularly as the software evolves or changes.
	Black-box testing suitability	Indicates whether the testing technique can be effectively applied without access to internal source code details, relying exclusively on external interfaces or behavioral nalysis.

The first three criteria relate to source code under test characteristics which indirectly include coding language specific aspects, project requirements, domain constraints, and typical use cases for the tested software.

- **The control-flow complexity**

It is measured using different metrics, one of them is cyclomatic complexity, which is defined as number of linearly independent execution paths of the module under test, corresponding to number of conditional statements in the source code (e.g., branches). The control-flow graph (CFG) approach [7] using CFG that consists of nodes (N) as conditional statements and edges (E) as execution paths between them. Thus, cyclomatic complexity is calculated as $C(G) = E - N + 2$. A higher value indicates that more complex testing for the module is required.

- **The data-flow complexity**

It has no universally accepted method of measurement and is instead evaluated as a combination of analytical approaches.

The definition-usage approach analyses pairs of definition and usage of variables through the source code and corresponding variable interactions [8]. The complexity grows with number of definitions, re-definitions and interactions of variables as shown, Fig 1.

```
1. int x = 10;          // definition
2. if (y > 0) {
3.    x = y + 5;        // re-definition
4. }
5. printf("%d", x);    // usage
```

Fig. 1. Def-Usage data-flow example

The data-flow coverage metrics analyze the test coverage output to check if every variable definition executed at least once (is reachable).

The fan-in and fan-out metrics analyze and compare the number of dependencies entering and leaving the module respectively, giving information about coupling and data exchange complexity.

- **The semantical-rich input**

The semantically legal generation ability of the testing tool is crucial for the libraries from specific domains, in which not correctly generated input leads to low coverage.

Techniques

- **Symbolic execution** exploration mechanism of control paths treats input variables as symbolic not as concrete values. This method systematically enumerates possible execution paths and generates symbolic constraints to represent these flows. As for **concolic testing**, it complements symbolic execution by executing programs with concrete inputs while simultaneously generating symbolic constraints. Such approach mitigates certain scalability issues associated with purely symbolic approaches. One of the classic technique usage example is KLEE [9]**.** It was used to achieve high coverage in testing of GNU COREUTILS and BUSYBOX that significantly improved fault detection by systematically exploring multiple execution paths. The results show precise fault localization and improved software reliability.

- **Search-based testing** utilizes natural selection concepts to iteratively evolve test inputs toward optimal

solutions defined by a fitness function. The fitness function commonly measures objectives such as code coverage, fault detection rates, or execution performance. The EvoSuite [10] automatically generates effective unit test suites for Java applications, improving code coverage and fault detection capability through the genetic evolution of test inputs.

- **Reinforcement learning (RL)** dynamically generates and refines test inputs through iterative interactions with the software system. The process is guided by reward function that reflects coverage metrics or fault detection metrics. The strategy of RL algorithms is to adjust input generation ability over time to maximize long-term rewards. The DRIFT tool [11] applies reinforcement learning to dynamically generate test cases. It significantly improves coverage and fault discovery efficiency, particularly in complex software systems. Tests within Windows 10 were performed to justify the efficiency of the method.
- **Model-based testing** generates test cases systematically from formal or semi-formal models of the software behavior, which capture specifications or functional descriptions. In addition, **property-based testing** complements such results using technique of validation software behavior against predefined properties or invariants. It automatically generates input scenarios to falsify these properties. The SpecExplorer [12] generated exhaustive test cases for verifying protocol implementations, which ensures compliance with detailed system specifications. The QuickCheck [13] systematically validates software behavior against specified properties, while Randoop [14] generates valid method call sequences that effectively uncover functional faults in software APIs, significantly enhancing black-box testing effectiveness.
- **Mutation Testing** assesses the quality and effectiveness of test suites using intentionally injecting small artificial faults (mutations) into the source code and observing whether existing test cases can detect these introduced faults. The PIT [15] has been used to assess and improve test suites in Java applications by injecting faults, leading to a significant increase in the robustness and quality of test cases. The Table 2 presents a comparison of the techniques based on their coverage effectiveness with respect to the characteristics of the tested source code.

Table 2 – **Comparison of techniques based on source code characteristics criteria**

Techniques	Control Flow Complexity	Data Flow Complexity	Semantical Rich Input
Symbolic execution	High	High	Low
Search-Based Testing	High	Medium	Medium
Reinforcement Learning	High	Medium	Medium
Model-Based Testing	High	High	High
Mutation Testing	Medium	Medium	Low

These capabilities are rated as High, Medium, Low respectively. The model-based testing technique receives a high rank in all three criteria. However, the comparison doesn't account for its automation potential, which is constrained by the need for manual settings and model creation limitations. To address this issue three operational criteria classification has been done. The comparison on operational properties shown on Fig. 2.

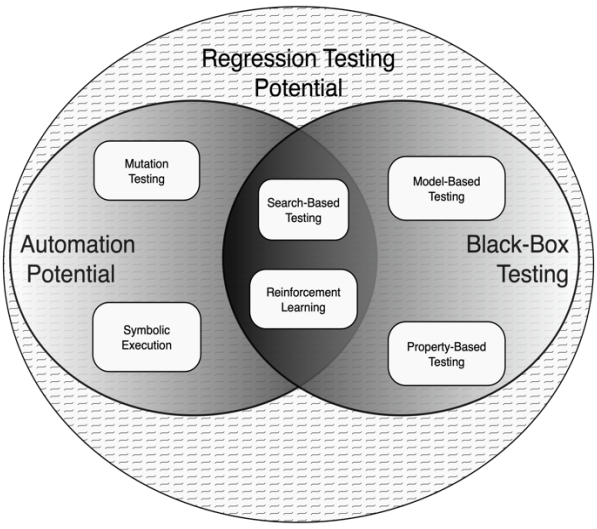

Fig. 2/ The techniques' classification depending on testing tool operational characteristics

Among all considered techniques, search-based testing emerges as the most appropriate candidate for enhancing CIDER. It aligns perfectly with CIDER's black-box nature, fully supports automation, and can replace current heuristic strategies with more adaptive and scalable optimizations. RL is also highly compatible, offering dynamic learning of effective test scenarios through interaction, ideal for evolving systems. In contrast, techniques like symbolic execution and mutation testing require source code access and are thus incompatible with CIDER's concept.

Investigation

Problem 1: Script input optimization.

Formulation. Let S denote a fixed test script consisting of a predefined sequence of function calls from a finite set of operations $\mathcal{F}$. Each invocation within S requires a vector of input parameters $x = (x_1, x_2, \dots, x_m), m \in \mathbb{N}$. Executing the script S with a given input vector x yields a dynamic execution trace, from which a coverage metric $C\big(S(x)\big) \in \mathbb{R}$ is computed. The coverage function quantifies the extent to which the input-triggered execution explores the program under test, based on structural criteria such as line, branch, or function coverage.

Objective. The objective is to identify the input vector x' that maximizes the achieved code coverage.

Investigation. The possible directions of search-based testing algorithms usage are:

- Artificial Bee Colony (ABC)
- Firefly algorithm (FA)
- Cuckoo Search (CS)

A comparison of the algorithms based on their advantages and drawbacks is shown in Table 3.

Table 3 – **Comparison of search-based metaheuristic algorithms**

	Advantages	Disadvantages
ABC	Provides a well-balanced exploration and exploitation mechanism.	Exhibit slower convergence on complex landscapes.
FA	Rapid convergence in continuous domains. Highly effective for unimodal problems.	Low performance in discrete spaces. Risk of being trapped in local optima.
CS	Highly effective for multimodal landscapes. Requires minimal parameter tuning.	May converge prematurely; less interpretable in terms of search behavior.
ACO	Highly effective for sequential and graph-based optimization. Adapts well to problem structure.	Slower convergence and requires more elaborate parameter tuning.

In addition, common algorithm of metaheuristic search optimization is shown in Fig. 3, with basic input parameters as fitness function C, number of iterations T.

Metaheuristic Search Optimization

Initialize $x = (x_1, x_2, \dots, x_m)$
Evaluate $C(S(x_i))$ **for each** x_i
$x_{best} = argmax(C(S(x_i)))$
for $t = 1$ to T **do**
  for each solution x_i **do**
    Generate $x_i' \leftarrow Gen(x_i, x_{best}, t, \dots)$
    Evaluate $C(S(x_i'))$
    if Acceptable (x_i') **then**
      $x_i \leftarrow x_i'$
      if $C(S(x_i)) > C(S(x_{best}))$ **then**
        $x_{best} \leftarrow x_i$
      end if
    end if
  end for
  Optionally Diversify (x)
end for
return x_{best}

Fig. 3 Generic representation of metaheuristic search optimization algorithm

▪ **Artificial Bee Colony**. The algorithm was introduced by D. Karaboga [16] and inspired by behavior of honeybee colonies. Three kinds of branches of logic are included: employed bees, onlooker bees and scouts. The employed bee searches on m iteration the solution vector x'_{mi} according to formula:

$$x'_{mi} = x_{mi} + \varphi_{mi}(x_{mi} - x_{ki}), \quad (1)$$

where x_{mi} – current solution, $\varphi_{mi} \in [-1,1]$ – the random number, x_{ki} – randomly chosen solution in memory.

The employed bees share their food source information with onlooker bees which choose a food source depending on the probability values, calculated as:

$$P_m = {C_m(x_{mi})}\Big/{\sum_i C_m(x_{mi})}. \quad (2)$$

After the x_{mi} is chosen it's modified via eq. (1). The employed bees whose solutions failed to improve for limit times become the scouts. The scout bee initializes their solution randomly.

▪ **Firefly algorithm.** The algorithm was firstly introduced in 2009 [17] and is inspired by fireflies' behavioral attraction to bright light. The search iteration at m can be described as:

$$x'_{mi} = x_{mi} + \beta(x_{mi} - x_{ki}) + \alpha\varphi_{mi}, \quad (3)$$

where x_{mi} – firefly's current solution, $\varphi_{mi} \in [-1,1]$ is the random number, α – scale factor, x_{ki} – randomly chosen solution in memory.

The attraction β from eq. (3) is as follows:

$$\beta = \beta_0 e^{-\gamma r_{mi}^2}, \quad (4)$$

where γ – absorption coefficient, r_{mi}^2 – distance between fireflies, β_0 – the attractiveness at distance 0.

The distance can be calculated as Cartesian or any other interpretation.

▪ **Cuckoo search.** The generation of new solutions behaves as cuckoo which lays own eggs in neighbor's' nest [18]. The solution search formula incorporates Lévy distribution:

$$x'_{mi} = x_{mi} + \text{Lévy}(\lambda). \quad (5)$$

The Lévy distribution is as follows:

$$\text{Lévy} \sim u = t^{-\lambda}, \quad (6)$$

where $\lambda \in (1; 3]$. The best solutions are kept for the next iteration and worse are dropped by the nest owner with probability P.

Problem 2: Call sequence optimization.

Let $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ denote the finite set of available function calls provided by a software under test interface. A test script is defined as an ordered sequence of function invocations, denoted $S = \{F_{i1}, F_{i2}, \dots, F_{ik}\}$, where each $F_{ik} \in \mathcal{F}$, and the sequence length k may be fixed or subject to constraints reflecting practical limitations on test depth or execution time.

The execution of a script S produces a runtime behavior of the program under test, resulting in a dynamic trace from which a structural coverage metric $C(S(x)) \in \mathbb{R}$ is computed. This metric may correspond to branch coverage, path coverage, state coverage, or other relevant measures of behavioral exploration.

Objective. The objective is to determine the script $S' \in \mathcal{S}$ that maximizes the achieved coverage, where $\mathcal{S}$ is the space of syntactically and semantically valid call sequences.

Investigation. The possible directions of search-based testing algorithms usage are:

▪ **Ant Colony Optimization.** The algorithm is inspired by foraging behavior of ants [19]. The ants use pheromones to mark good paths between nest and food sources.

▪ Over time, shorter paths receive more pheromone, reinforcing them based on probability.

The problem can be modeled as a directed fully connected graph $G = (V, E)$, where $V \subseteq \mathcal{F}$ is the node representing method call, $E \subseteq V \times V$ are edges representing transitions between method calls - a path of

the graph represent searched call sequence S. Each edge has associated pheromone value τ_{ij} and a heuristic value η_{ij} representing desirability of the edge. Then each ant k in the state i (current method call) choses the next method j based on:

$$P_{ij}^{(k)} = {(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}\Big/{\sum_{l\in\mathcal{N}_i^k}(\tau_{il})^{\alpha}(\eta_{il})^{\beta}}, \quad (7)$$

where $\mathcal{N}_i^k$- unvisited neighbors of node i for ant k, and α, β – balancing coefficients.

After construction of S the reward R_k is computed and pheromone matrix is updated using scaling factor Q and call sequence length L_k:

$$\tau'_{ij} = {Q \cdot R_k}\Big/{L_k}. \quad (8)$$

The algorithm is stopped when best sequence is found, or time budget is over.

The next papers [20, 21] have deeply dived into experimental application of ACO optimization for method call sequences generation in test scenarios. Their results showed that ACO generated sequences achieved higher code coverage and robustness compared to random or greedy baselines, particularly in preserving object usage patterns and handling class constructors correctly.

In addition to graph-based metaheuristic techniques such as ACO, RL provides a learning-based approach to generating call sequences, particularly Q-learning.

- **Q-learning.** The algorithm was introduced by C. Watkins [22] and is a form of model-free RL.

The generic algorithm of learning phase is shown on Fig. 4, where:

s – state, representing partial sequence of function call,

a – action selecting next valid and capable method from $\mathcal{F}$,

r – reward, a scalar value based on $C(s)$, and the return represents policy,

$\pi(s)$ – which maximizes the cumulative reward for doing action a in state s.

The update of Q-Table phase is done using formula:

$$Q'(s,a) \leftarrow Q(s,a) + \Delta Q(s',a'),$$

$$\Delta Q(s',a') = \alpha \cdot [r + \gamma \cdot max_{a'} Q(s',a') - Q(s,a)],$$

where $\alpha \in (0;1]$ – learning rate, γ – discount factor.

Q-learning Algorithm

```
Initialize Q-Table Q(s, a) ← 0
for each episode do
  Initialize State (s)
  for t = 1 to T do
  if Exploration (ε) then
    a = Random(A)
  else Exploitation
    a = argmax_a Q(s, a)
  end if
  s' = Step(s, a)
  r = GetReward(s')
  Update Q-Table (s', s, r)
  if TerminalState (s')
    break
  s ← s'
  end for
end for each
return π(s) = argmax_a Q(s, a)
```

Fig. 4. Generic representation of learning section of Q-learning algorithm

Conclusions

This paper presents a comprehensive overview of automated test generation techniques applicable to C++ software, with a particular focus on enhancing the capabilities of the CIDER tool.

The search-based and reinforcement learning techniques offer adaptive, scalable, and fully automated mechanisms for high-coverage test scenarios generation. In particular, search-based approaches such as Artificial Bee Colony, Firefly Algorithm, and Cuckoo Search show strong potential in test inputs optimization. On the other hand, Ant Colony Optimization has effectiveness in method call sequence generation. Reinforcement learning complements the approach via dynamic learning from test execution feedback, thereby improving test effectiveness over time. Symbolic and concolic execution, despite their high precision, are constrained by CIDER's lack of source code access, making them less suitable in this context. The mutation testing, although useful for evaluating test suite robustness, offers limited applicability for full process automation. In conclusion, integration of adaptive metaheuristics and learning-based optimization techniques into CIDER's architecture shows promising results in improvement of test coverage and full automation potential.

REFERENCES

1. Zhang, Y., Lu, Q., Liu, K., Dou, W., Zhu, J., Qian, L., Zhang, C., Lin, Z., & Wei, J. (2025). CITYWALK: Enhancing LLM-Based C++ Unit Test Generation via Project-Dependency Awareness and Language-Specific Knowledge. *arXiv preprint arXiv:2501.16155*. https://doi.org/10.48550/arXiv.2501.16155
2. Rho, S., Martens, P., Shin, S., Kim, Y., Heo, H., & Oh, S. (2023). Coyote C++: An Industrial-Strength Fully Automated Unit Testing Tool. *arXiv preprint arXiv:2310.14500*. https://doi.org/10.1016/j.jss.2022.111259
3. Wang, Y., Mäntylä, M. V., Liu, Z., & Markkula, J. (2022). Test automation maturity improves product quality - Quantitative study of open source projects using continuous integration. *Journal of Systems and Software*, 188, 111259. https://doi.org/10.1016/j.jss.2022.111259
4. Feloney, S. (2023). *State of Test Automation: Trends and Priorities for 2023*. DevOps Digest. https://www.devopsdigest.com/state-of-test-automation-2023
5. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., & Harman, M. (2018). Mutation testing advances: An analysis and survey. *Advances in Computers*, 112, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015
6. Hulevych, M. (2024). CIDER: Assisted automation tool for C++ libraries testing. *Control, Navigation and Communication Systems*, 2(76), 74–77. https://doi.org/10.26906/sunz.2024.2.074
7. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

8. Rapps, S., & Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), 367–375. https://doi.org/10.1109/TSE.1985.232226
9. Cadar, C., Dunbar, D., & Engler, D. (2008). KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, San Diego, CA, USA. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf
10. Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, Szeged, Hungary, 416–419. https://doi.org/10.1145/2025113.2025179
11. Harries, L., Storan Clarke, R., Chapman, T., Nallamalli, S. V. P. L. N., Ozgur, L., Jain, S., Leung, A., Lim, S., Dietrich, A., Hernández-Lobato, J. M., Ellis, T., Zhang, C., & Ciosek, K. (2020). Drift: Deep reinforcement learning for functional software testing. *arXiv preprint arXiv:2007.08220*. https://doi.org/10.48550/arXiv.2007.08220
12. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., & Nachmanson, L. (2008). Model-based testing of object-oriented reactive systems with Spec Explorer. In R. M. Hierons, J. P. Bowen, & M. Harman (Eds.), *Formal methods and testing* (pp. 39–76). Springer. https://doi.org/10.1007/978-3-540-78917-8_2
13. Claessen, K., & Hughes, J. (2011). QuickCheck: A lightweight tool for random testing of Haskell programs. *SIGPLAN Notices*, 46(4), 53–64. https://doi.org/10.1145/1988042.1988046
14. Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation. *Proc. 29th Int. Conf. on Software Engineering (ICSE '07)*, Minneapolis, MN, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37
15. Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016). PIT: A practical mutation testing tool for Java (Demo). *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, Saarbrücken, Germany, 449–452. https://doi.org/10.1145/2931037.2948707
16. Karaboga, D. (2010). Artificial bee colony algorithm. *Scholarpedia*, 5(3), 6915. https://doi.org/10.4249/scholarpedia.6915
17. Yang, X.-S. (2009). Firefly algorithms for multimodal optimization. In O. Watanabe & T. Zeugmann (Eds.), *Stochastic algorithms: Foundations and applications. SAGA 2009, Sapporo, Japan, October 26–28, 2009* (pp. 169–178). Lecture Notes in Computer Science (Vol. 5792). Springer. https://doi.org/10.1007/978-3-642-04944-6_14
18. Yang, X.-S., & Deb, S. (2009). Cuckoo search via Lévy flights. *In Proceedings of the World Congress on Nature & Biologically Inspired Computing (NaBIC 2009)*, 210–214. IEEE. https://doi.org/10.1109/NABIC.2009.5393690
19. Dorigo, M., Maniezzo, V., & Colorni, A. (1996). The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1), 29–41. https://doi.org/10.1109/3477.484436
20. Nayak, G., & Ray, M. (2022). Model-based test sequence generation and prioritization using ant colony optimization. *Journal of Information Technology Research*, 15(1), 1–17. https://doi.org/10.4018/JITR.299946
21. Palak, P., & Gulia, P. (2018). Ant Colony Optimization Based Test Case Selection for Component Based Software. International Journal of Engineering and Technology, 7(4), 2743-2745. https://doi.org/10.14419/ijet.v7i4.17565
22. Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3–4), 279–292. https://doi.org/10.1007/BF00992698

Received (Надійшла) 18.03.2025
Accepted for publication (Прийнята до друку) 14.05.2025

ВІДОМОСТІ ПРО АВТОРІВ / ABOUT THE AUTHORS

Гулевич Михайло Володимирович – аспірант кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Mykhailo Hulevych – PhD student, Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: gulevich30misha@gmail.com; ORCID Author ID: https://orcid.org/0009-0003-8622-3271.

Коломійцев Олексій Володимирович – доктор технічних наук, професор, професор кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Oleksii Kolomiitsev – Doctor of Technical Sciences, Professor, Professor of the Computer Engineering and Programming Department, National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine;
e-mail: Oleksii.Kolomiitsev@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0001-8228-8404;
Scopus ID: https://www.scopus.com/authid/detail.uri?authorId=57211278112.

Методики автоматизації генерації тестів для програмного забезпечення C++

М. В. Гулевич, О. В. Коломійцев

Анотація. Автоматизація тестування є критично важливою для забезпечення якості програмного забезпечення. Автоматизовані підходи значно скорочують зусилля на тестування, покращують ефективність тестування та підвищують загальну надійність програмного забезпечення. Інструмент CIDER, представлений автором, пропонує багатообіцяюче автоматизоване рішення шляхом генерації тестових сценаріїв на основі запису виконання програм і застосування евристичної пошукової оптимізації, зокрема гармонічного пошуку. Незважаючи на свої переваги, CIDER стикається з обмеженнями, включаючи неповне охоплення складної логіки розгалуження, неефективне дослідження великих просторів вхідних даних, труднощі в обробці семантично насичених або контекстуально складних вхідних даних. Ця оглядова стаття спрямована на систематичне дослідження, порівняння та оцінку різних автоматизованих методик генерації тестів, таких як символічне виконання, конколічне тестування, еволюційні алгоритми, навчання з підкріпленням і тестування на основі моделі. **Мета** полягає в тому, щоб класифікувати ці методи на основі їхніх характеристик, оцінити їхню ефективність у вирішенні наявних недоліків CIDER та визначити відповідні підходи, які могли б розширити можливості інструменту для автоматизованої генерації тестів з точки зору ефективності покриття коду.

Ключові слова: автоматизація тестування, C++, регресійне тестування, бджолиний алгоритм, алгоритм мурашиної оптимізації, алгоритм світлячків, пошук зозулі, навчання за q-функцією.