The Name of the Title Is Hope

ZHENG LI*, Southeast University, China

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:

Zheng Li. 2018. The Name of the Title Is Hope. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

This chapter will detail how to leverage artificial intelligence techniques in software testing, specifically focusing on optimization methods based on Boundary Coverage Distance (BCD) and techniques using Reinforcement Learning (RL), to achieve intelligent test case generation and boundary exploration. First, in the introduction, we present the background and significance of this chapter. Then, this section is divided into two core subsections, which respectively introduce BCD-optimized boundary value analysis and the application of reinforcement learning in test exploration. Finally, we summarize and prospect the content of this chapter.

As software systems become increasingly complex and the requirements for reliability and safety continue to rise, traditional test case design methods that rely on manual experience can no longer meet practical needs. On one hand, traditional equivalence class partitioning and Boundary Value Analysis (BVA) methods, when facing high-dimensional and multi-branch complex software, require manual extraction of a large number of boundary conditions, which is extremely challenging and inefficient. On the other hand, Random Testing (RT) and Adaptive Random Testing (ART), although able to improve test coverage to some extent, still lack exploration capability for boundary defects. Therefore, how to maximize the coverage of potential boundary defect areas in software while ensuring test efficiency has become a pressing challenge in the field of software engineering.

*Both authors contributed equally to this research.

Author's Contact Information: Zheng Li, Southeast University, Nanjing, China, LiZheng040910@163.com.

Manuscript submitted to ACM

AI-driven test case optimization emerges in this context. On one hand, by quantifying the boundary coverage with a metric like Boundary Coverage Distance (BCD), test case design can be transformed into an optimization problem, allowing intelligent algorithms to automatically search for optimal test points in the input domain. On the other hand, intelligent techniques such as reinforcement learning can dynamically construct environments and testing strategies, interacting with the system under test to efficiently approach failure boundaries and uncover defects in extreme scenarios. Combining both approaches not only reduces the manual workload of testers but also achieves more comprehensive and precise boundary testing in high-dimensional complex scenarios.

This chapter aims to clarify the principles and application effects of two main AI-driven methods for test case generation and boundary exploration. First, we will delve into BCD-based boundary value analysis, including the mathematical definition of BCD, its computation method, and the implementation of test case auto-generation using MCMC (Markov Chain Monte Carlo) optimization strategies. Then, considering dynamic scenarios such as autonomous driving, we will discuss the advantages of reinforcement learning in test exploration, emphasizing the necessity of dynamic threshold design in the boundary approach process. By citing real experimental results and visualization examples from the literature, we demonstrate the significant improvements in defect detection rates and efficiency provided by these two approaches. Finally, we offer a summary of this chapter and look forward to future integrations of AI and software testing.

3.1 Boundary Value Analysis with BCD Optimization

Before formally introducing BCD, let us first review the basic idea of Boundary Value Analysis (BVA). BVA is a black-box testing technique based on the experience that "system defects often occur near boundary conditions. Its core is to select values near partition boundaries in the input domain to construct test cases, aiming to identify "off-by-one errors or "boundary omissions." However, traditional BVA often relies on testers manually identifying equivalence partitions and corresponding boundaries based on requirement documents. For software systems with high-dimensional input domains and complex constraint relationships, this manual approach is time-consuming and prone to omissions.

To address the insufficient coverage by manually selecting boundary values, Guo *et al.*[1] proposed a metric called **Boundary Coverage Distance (BCD)** and, based on this, designed a test case optimization algorithm using MCMC. Compared to traditional BVA, which only focuses on "whether a boundary is hit," BCD introduces an integer distance metric that measures the minimum distance between test cases and all boundary points, thus transforming the boundary coverage problem into an optimization problem. This metric can reflect the distribution of the test suite across the entire boundary space and supports the automated generation of optimal test cases.

First, we provide the mathematical definitions of equivalence partitions and boundaries. Let the input domain of the software under test be I, and the output domain be $O = f(I)$. We divide the output into m mutually disjoint categories: $O = \bigcup_{i=1}^{m} O_i$, with $O_i \cap O_j = \emptyset$ for $i \neq j$. Correspondingly, define the input equivalence partitions:

$$I_i := \{ x \in I \mid f(x) \in O_i \}, \quad i = 1, 2, \ldots, m. \tag{1}$$

Any input $x \in I$ yields an output that belongs to some category O_i, so x belongs to the corresponding equivalence partition I_i. Within this framework, we need to clarify the minimal unit of "input change." Let G be a set of functions satisfying the following properties:

(1) For any $g \in G$, its inverse operation g^{-1} also belongs to G.
(2) For any $x, y \in I$, there exists a finite sequence of operations $\{g_1, g_2, \ldots, g_n\} \subset G$ such that $y = g_1 \circ g_2 \circ \cdots \circ g_n(x)$.

A function $g \in G$ is called a "minimal operation" on the input, and having its inverse ensures reversibility. Then, define the boundary set of the equivalence partition I_i as:

$$B_i := \{ x \in I_i \mid \exists\, g \in G,\ f(g(x)) \notin O_i \}. \tag{2}$$

In other words, if an input x belongs to I_i, but applying a minimal operation to it causes it to leave the equivalence partition I_i, then x is considered a boundary point. Geometrically, B_i lies on the boundary between I_i and other equivalence partitions.

Next, we introduce the distance measure between test inputs and boundary points. Let $d(x, y)$ denote the number of minimal operations needed to go from input x to input y, i.e.,

$$d(x, y) \ = \ \min\{ n \geq 0 \mid y = g_n \circ g_{n-1} \circ \cdots \circ g_1(x),\ g_j \in G \}. \tag{3}$$

For example, in the "English exam grading" example, if $G = \{ g_1 : \text{Listening} + 1,\ g_2 : \text{Listening} - 1,\ g_3 : \text{Reading} + 1,\ g_4 : \text{Reading} - 1 \}$, then going from input $x = (-1, -1)$ to $y = (0, 0)$ requires two steps (apply g_1 then g_3), so $d((-1, -1), (0, 0)) = 2$.

Let the test set for an equivalence partition I_i be $T_i \subseteq I_i$. To measure how well T_i covers its boundary B_i, we compute for each boundary point $y \in B_i$ the minimum distance to any test point in T_i and then take the maximum over all y:

$$d(T_i,\ B_i) \ = \ \max_{y \in B_i} \min_{x \in T_i} d(x, y). \tag{4}$$

If T_i is empty, define $d(T_i, B_i) = +\infty$. Intuitively, equation (4) first finds the closest test point to each boundary point, then the worst-case distance over all boundary points, reflecting the "weakest" boundary coverage. Finally, let the overall test set be $T = \bigcup_{i=1}^{m} T_i$ and the union of all boundary sets $B = \bigcup_{i=1}^{m} B_i$. We define the overall boundary coverage distance as:

$$\mathrm{BCD}(T) \ = \ d(T, B) \ = \ \max_{i=1,\ldots,m} d(T_i,\ B_i). \tag{5}$$

When $\mathrm{BCD}(T) = 0$, the test set T covers every boundary point in all partitions. Conversely, if $\mathrm{BCD}(T) = k > 0$, there exists some partition whose boundary point is at distance k from the nearest test point, indicating incomplete boundary coverage.

With BCD as a metric, test case generation becomes a *BCD minimization* problem: given a fixed number of test cases n, how do we select or optimize an initial test set T to minimize $\mathrm{BCD}(T)$? To solve this optimization problem, Guo *et al.*[1] borrow ideas from the Markov Chain Monte Carlo (MCMC) algorithm and propose three main test input generation and optimization strategies, referred to as **Algorithm 1**, **Algorithm 2**, and **Algorithm 3**. Below is a brief introduction to the core ideas and implementation steps of these algorithms.

Algorithm 1: Greedy Descent BCD Optimization. Algorithm 1 first *randomly* generates n initial test points in the input domain, forming the set $T = \{t_1, t_2, \ldots, t_n\}$. It then iteratively executes the following steps until a preset iteration count is reached or the BCD value converges:

(1) Compute the current BCD value of the test set T, i.e., $\mathrm{BCD}(T)$.
(2) Randomly select a test point t from T, and generate a candidate input t' according to a *proposal distribution*. Typically, the proposal distribution perturbs t by adding random changes such as $\{-1, 0, 1\}$ for numerical data; for discrete data, switch to a neighboring state.

(3) Form the candidate test set T', where t is replaced by t'. Compute BCD(T'). If BCD(T') < BCD(T), **accept** the candidate and update $T \leftarrow T'$; otherwise, **reject** it and keep T unchanged.

By repeatedly iterating, the test points gradually shift from a random distribution toward an optimal distribution that spans the boundaries. Because the algorithm only accepts candidates that strictly reduce BCD, Algorithm 1 is essentially a *greedy descent* strategy and can easily become trapped in local optima. To illustrate this process, it is analogous to the Metropolis-Hastings acceptance rule:

$$P_{\text{accept}} = \begin{cases} 1, & \text{BCD}(T') < \text{BCD}(T), \\ 0, & \text{otherwise.} \end{cases}$$

Although simple, this algorithm has been shown in practice to significantly improve boundary coverage of the test set. In the experiment with the English exam grading program, the initial random set of 30 test points had a large BCD. After 10 000 iterations, all test points approached the boundary lines, resulting in BCD(T) = 0 (meaning every boundary in each partition was directly hit). Correspondingly, the mutation kill rate improved from 36% for random testing to over 80%.

Algorithm 2: "Harmless" Boundary Improvement Strategy. Because Algorithm 1 is too strict—only accepting candidates that reduce BCD—it can easily get stuck in local optima. Algorithm 2 refines the acceptance criterion by introducing the notion of a "*harmless improvement*": if a candidate reduces the distance to at least one boundary point and does not increase the distance to any other boundary point, then accept the candidate; otherwise, reject it. Formally:

(1) Select the current test set T, randomly pick a point $t \in T$, and generate a candidate t'.
(2) For each partition I_i, compare the distance of each boundary point $y \in B_i$ before and after replacement. Let b_{decrease} be the number of boundary points whose distance decreases under the candidate, and b_{increase} be the number of boundary points whose distance increases.
(3) If $b_{\text{decrease}} > 0$ and $b_{\text{increase}} = 0$, then accept the candidate ($T \leftarrow T'$); otherwise, reject it.

Since this criterion allows candidates as long as they improve at least one boundary without harming others, it has greater exploratory power than Algorithm 1. Experiments show Algorithm 2 can quickly reduce BCD in the early stages, but later, because it only permits "non-degrading improvements," it may stagnate in some partitions. However, compared to Algorithm 1, Algorithm 2 shows a slight improvement in mutation kill rates for most benchmark programs. For example, on the triType program, Algorithm 1 under the BCD_{max} metric achieved an 80% kill rate, while Algorithm 2 improved it to 85%.

Algorithm 3: Probabilistic Acceptance (MCMC with Temperature). Algorithm 3 further introduces a *probabilistic acceptance* mechanism, allowing a candidate to be accepted with some probability even if it increases certain boundary distances, thus enabling escape from local optima. The detailed steps are:

(1) As in Algorithm 2, generate the candidate set T' and compute b_{decrease} and b_{increase}.
(2) If $b_{\text{decrease}} > 0$ and $b_{\text{increase}} = 0$, accept T' unconditionally.
(3) Otherwise, accept T' with probability

$$P_{\text{accept}} = \exp\big(-(b_{\text{increase}} - b_{\text{decrease}}) / T\big),$$

where T is the temperature parameter in simulated annealing, which gradually decreases over iterations. If not accepted, reject the candidate.

This strategy balances exploration and exploitation. In the early high-temperature phase, it allows more aggressive exploration by accepting some candidates that slightly increase BCD; as the temperature lowers, only a small number of "weakly degrading" candidates are accepted, eventually converging to a better solution. Experimental results demonstrate that Algorithm 3 outperforms Algorithm 2 in avoiding early convergence to local optima, especially in complex programs with uneven boundary distributions (e.g., miniSAT). For instance, in the miniSAT test experiment, Algorithm 2 under BCD_i achieved approximately an 85% kill rate, whereas Algorithm 3 under BCD_i improved it to 93%.

Boundary Distribution for English Exam Grading. Figure 1 shows the basic boundary distribution of the English exam grading program on the two-dimensional input plane. In this figure, the gray region represents invalid inputs (I_1), the blue region represents "fail" (I_3), and the orange region represents "pass" (I_2). The dashed black lines depict the equivalence partition boundaries as defined by equations (3)-(5). Each boundary corresponds to inputs where a minimal operation can move a score combination from one output category to another. This illustration provides a clear view of how the input domain is partitioned into I_1, I_2, and I_3 and where their shared boundaries lie.

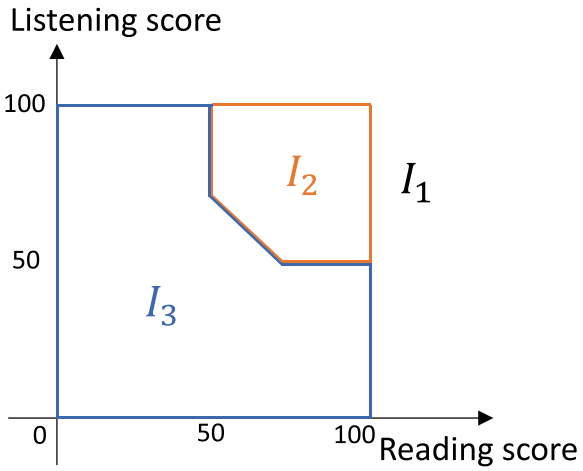

Fig. 1. Basic boundary distribution for the English exam grading program: gray indicates invalid inputs (I_1), blue indicates fail (I_3), orange indicates pass (I_2), and black dashed lines show the equivalence partition boundaries.

In the BCD metric experiment, the random test set of 30 points had BCD = 5 (meaning some boundary points were five minimal steps away from the nearest test points), while Algorithm 1 under the $\mathrm{BCD}_{\mathrm{mean}}$ criterion reduced BCD to 0 after 10 000 iterations, achieving direct coverage of all boundary points. Correspondingly, the mutation kill rate improved from 36% (random testing) to 80%. This experimental result fully demonstrates the effectiveness of BCD-based boundary value optimization in black-box testing.

3.2 Reinforcement Learning for Test Exploration

Traditional static boundary value analysis methods have inherent limitations when dealing with **dynamic decision systems**. Take an autonomous driving system as an example: test scenarios are not limited to single-input judgments but involve continuous, multi-round interactions between the system and the environment. Such scenarios typically involve continuous state spaces, high-dimensional action spaces, and uncertain environmental disturbances. Therefore,

efficiently generating test sequences that can trigger potential faults or extreme behaviors is challenging. To address this, researchers have introduced **reinforcement learning** techniques into the domain of software testing. Through interactions between an agent (the RL tester) and the system under test in a simulated environment, the agent actively explores and approaches the system's failure boundary.

Reinforcement learning-driven test exploration can be broken down into the following core elements: (1) **state representation**, which abstracts the current system and environment status into features recognizable by the RL agent; (2) **action space**, representing the controllable interventions the testing agent can apply to the environment or the system under test; (3) **reward function**, which measures whether the sequence of actions successfully approaches or triggers the failure boundary; and (4) **policy learning**, whereby the agent continuously interacts with the environment to optimize its policy to achieve the desired goal within a limited number of steps. In the context of autonomous driving testing, the state might include current vehicle speed, distance to the car in front, and road type; the agent's actions could be accelerate, brake, steer, or introduce environmental disturbances; and the reward function is typically designed as the negative distance to the safety boundary (closer distances receive higher reward), or if a collision occurs, a large positive reward is given to encourage exploration of dangerous scenarios.

Agent-Environment Construction. For an autonomous driving control system under test, we can construct the reinforcement learning environment as follows:

- **Environment**: A simulated road scenario, including lanes, curb, obstacles, pedestrians, and other traffic participants. This environment can be built using a high-fidelity simulator (e.g., CARLA, Gazebo) that provides real-time vehicle dynamics models and sensor data feedback.
- **State (s_t)**: A vector representing the system information at timestep t, typically including vehicle speed v_t, distance to the car in front d_t, lane deviation θ_t, driver-set speed limit s_{limit}, and road condition information r_t.
- **Action (a_t)**: Defines the action the agent can take at timestep t, including setting control inputs for the vehicle under test (throttle, brake, steering angle) and introducing environmental perturbations (e.g., sudden lane change by an adjacent vehicle, change in road friction due to rain).
- **Reward (r_t)**: Guides the agent to approach the failure boundary. Common reward design patterns include:
  (1) *Negative safety distance reward*: If the current distance to the vehicle in front d_t minus the safety threshold d_{safe} yields $\Delta_t = d_t - d_{\text{safe}}$, then the smaller Δ_t is, the higher the reward; if $\Delta_t < 0$ (i.e., the boundary is approached or crossed), a collision is triggered, giving a large positive reward and ending the episode.
  (2) *Boundary-trigger reward*: If the agent drives the system without failure for several consecutive steps, maintain a small reward; if a collision or extreme event like sudden emergency braking occurs, give a one-time large positive reward to encourage the discovery of more extreme boundary scenarios.
  (3) *Dynamic threshold penalty*: Gradually lower the safety distance threshold d_{safe} during testing; if the agent can still avoid failure under a lower threshold, give a negative penalty, prompting the agent to find more stringent conditions.

Through this design, the RL agent can continuously explore questions like "How to approach the collision boundary?" and "How to identify the most challenging spacing that avoid collision among mixed traffic?" in the simulated environment, uncovering potential flaws at critical conditions.

Dynamic Threshold Design and Case Illustration. In reinforcement learning-based testing, introducing a **dynamic threshold** is key to effectively achieving "approaching the failure boundary while ensuring safety." A case study in [2]

(Game Theoretic Modeling of Driver and Vehicle Interactions for Verification and Validation of Autonomous Vehicle Control Systems) provides an enlightening approach. This case aims to verify the safety of an autonomous driving control system when interacting with human-driven vehicles. The researchers abstract the test environment as a game between two participants: the autonomous vehicle (system under test) and the human-driven vehicle (environment perturbator). Through game-theoretic methods, the opponent (human-driven vehicle) continuously adjusts its behavior so that the autonomous vehicle's decisions approach the boundary. The core ideas are:

(1) **Safety Channel Mechanism**: Design a dynamic safety threshold rule for the autonomous driving system. When sensors detect that the distance to the vehicle in front exceeds a threshold $d_{\max}$, there is no safety risk; but when the distance d_t approaches a threshold d_{th}, emergency braking is triggered; if a vehicle in the adjacent lane is speeding and approaching quickly, further reduce d_{th} to ensure a more conservative safety boundary. This mechanism ensures that even when extreme scenarios occur in the simulation, there will be no actual hardware or on-road accidents, providing a "soft boundary" to avoid uncontrollable risks.

(2) **Adaptive Thresholds**: During each test episode, the agent dynamically adjusts d_{th} based on environmental feedback (such as distance to the vehicle in front, relative speed of surrounding vehicles, and road conditions). Initially, d_{th} is set conservatively to ensure stable driving within this threshold. If no failures occur after multiple test rounds, d_{th} is gradually lowered to guide the autonomous system to operate under increasingly challenging conditions, thereby approaching the failure boundary and uncovering potential defects.

(3) **Environment Agent Modeling**: Model the human-driven vehicle as an agent that learns, via reinforcement learning or game-theoretic methods, how to challenge the autonomous vehicle's decision boundary through sudden lane changes or tailgating at high speed. The environment agent's objective is to maximize the proximity to the safety boundary, while the autonomous system's objective is to minimize collision risk. Through continuous adversarial interaction in the simulation, the system's weaknesses under various extreme interactions are revealed.

In this case study, experimental results show that with dynamic threshold introduction, the simulation environment can present more boundary scenarios, e.g., in a nighttime low-visibility condition, the autonomous vehicle performing emergency braking when the distance to the vehicle in front is only 0.5 meters. Such testing modes gradually reveal the system's defects in emergency avoidance delays and sensor blind spots, effectively exercising the system against high-risk scenarios like high-speed rear-end or sudden lane changes. Without dynamic thresholds and relying on a fixed threshold of $d_{\text{th}} = 2$ meters, it would not be possible to approach the true collision boundary, leaving many potential defects unexposed. Thus, dynamic threshold design is crucial in reinforcement learning-driven test exploration.

Reinforcement Learning Algorithm Example. In the aforementioned environment, one can use **Deep Reinforcement Learning (Deep RL)** algorithms such as Deep Deterministic Policy Gradient (DDPG) or Proximal Policy Optimization (PPO) to train the agent. Taking PPO as an example, the training process is as follows:

(1) Initialize the policy network $\pi_\theta(a_t \mid s_t)$ and value network $V_\phi(s_t)$ with random weights θ and ϕ.

(2) Execute τ steps in the simulation environment: at each step, choose an action a_t according to the current policy π_θ, receive the next state s_{t+1} and reward r_t, and store the transition (s_t, a_t, r_t, s_{t+1}).

(3) Compute the temporal difference target and advantage function:

$$A_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t),$$

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 V_\phi(s_{t+2}) + \cdots - V_\phi(s_t),$$

where $\gamma \in (0, 1)$ is the discount factor.

(4) Update the value network parameters ϕ by minimizing the loss:

$$L_\phi = \mathbb{E}_t\left[\left(V_\phi(s_t) - G_t\right)^2\right].$$

(5) Update the policy network parameters θ by maximizing the clipped probability ratio loss:

$$L_\theta = \mathbb{E}_t\Big[\min\big(r_t(\theta)\, A_t,\ \mathrm{clip}\big(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon\big)\, A_t\big)\Big],$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)},$$

and ϵ is the clipping hyperparameter (e.g., 0.2). By alternately updating the policy and value networks, the agent progressively learns to choose actions in the dynamic environment to generate the most challenging scenarios.

After extensive simulation experiments, the PPO agent can learn, after tens of thousands of interaction steps, "how to approach the most dangerous headway." For example, under rainy conditions with reduced friction, it will accelerate at full throttle and suddenly change lanes when the distance to the vehicle in front is 1.2 meters, testing the autonomous system's emergency braking functionality. As training continues and the threshold gradually decreases to 0.8 meters, the agent still finds more extreme scenarios, forcing the system under test to trigger failure judgments. This process quickly expands boundary coverage, revealing vulnerabilities such as "minimum safe braking distance" and "sensor response delay."

Experimental Results and Comparison. To evaluate the effectiveness of reinforcement learning-driven test exploration, related studies compare the defect detection performance of the following methods in an autonomous driving simulation scenario:

- **Fixed-Threshold Random Testing (Fixed-Threshold RT)**: Maintain a fixed safety distance threshold $d_{\text{safe}} = 1.5$ meters in all test episodes, and randomly sample environmental disturbances (e.g., random lane changes, random accelerations). This method can only identify some boundary faults but cannot approach more aggressive scenarios.
- **Fixed-Threshold Reinforcement Learning (Fixed-Threshold RL)**: Train the RL agent with a fixed threshold $d_{\text{safe}} = 1.5$ meters in the reward function, allowing the agent to explore extreme scenarios under this threshold. Compared to random testing, it can discover more subtle boundary faults, but being limited by the fixed threshold, it cannot continue exploration if d_{safe} needs to be further reduced.
- **Dynamic-Threshold Reinforcement Learning (Dynamic-Threshold RL)**: Adopt the aforementioned dynamic threshold design, where the agent iteratively updates the threshold from conservative to aggressive during training to approach the boundary. Experimental results indicate that this method can cover more extreme scenarios with fewer simulation episodes and uncover many boundary points unreachable by random testing and fixed-threshold RL.

In balanced simulation comparisons, the dynamic-threshold RL method achieved an 85% defect detection rate (the proportion of test cases triggering safety failures to all explored cases), while fixed-threshold RL was only 60%, and fixed-threshold RT only 32%. This result demonstrates that by introducing dynamic thresholds, the RL agent can explore more extreme boundary conditions while remaining safe, improving test efficiency.

This chapter has introduced two **AI-driven test case optimization and boundary exploration** methods: first, **BCD-based boundary value analysis**; and second, leveraging **reinforcement learning (RL)** for dynamic environment test exploration. The BCD method defines the mathematical metric BCD(T) to transform boundary coverage into an optimization objective, and combines MCMC strategies (greedy descent, harmless improvement, probabilistic acceptance) to generate optimal test sets. Experimental results show that BCD-optimized algorithms can significantly improve detection rates for boundary faults—for example, raising the mutation kill rate for the English exam grading program from 36% to over 80%. The reinforcement learning approach, aimed at dynamic decision systems such as autonomous driving, emphasizes the agent-system interaction in the simulation environment and employs carefully designed reward functions and dynamic thresholds to iteratively approach failure boundaries. Case studies and simulations demonstrate that dynamic-threshold RL discovers far more extreme scenarios than random testing or fixed-threshold RL, greatly enhancing defect detection.

By using AI-driven test case optimization, we shift from the paradigm of "manual observation + empirical selection" to "quantification + intelligent search," providing effective means to address testing challenges of contemporary complex software systems. Future work can proceed along the following directions: first, integrate BCD metrics with more diverse boundary representations (e.g., coverage of branch decision paths, data flow coverage) to further improve test quality; second, combine reinforcement learning with symbolic execution, fuzz testing, and other methods to achieve more comprehensive boundary exploration; third, build distributed simulation and parallel optimization platforms for industrial-scale software to enhance the usability and efficiency of AI-driven testing in real-world projects. Through continuous promotion of the deep integration of AI and software testing technologies, software system reliability and safety will be further improved.

References

[1] X. Guo, H. Okamura, T. Dohi. Optimal test case generation for boundary value analysis. *Software Quality Journal*, 32(2):543-566, 2024.

[2] N. Li, D. W. Oyler, M. Zhang, Y. Yildiz, I. Kolmanovsky, A. R. Girard. Game Theoretic Modeling of Driver and Vehicle Interactions for Verification and Validation of Autonomous Vehicle Control Systems. *IEEE Transactions on Control Systems Technology*, 26(5):1782-1797, 2018.

[3] T. Y. Chen, H. Leung, I. K. Mak. Adaptive Random Testing. In *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04)*, Springer, pp.320-329, 2004.

[4] S. Chib, E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327-335, 1995.