Automated Software Engineering (2026) 33:50
https://doi.org/10.1007/s10515-026-00598-w

Design and optimization of a test case generation algorithm for real-time embedded systems based on adaptive Q-Learning

Yingbei Niu[1] · **Soo See Chai**[1]

Received: 16 April 2025 / Accepted: 18 January 2026

Abstract
Testing real-time embedded systems requires intelligent strategies that balance test coverage, timing constraints, and resource limitations. The traditional test case generation methods, such as random testing and conventional Q-learning, often fail to adapt to dynamic workloads and maintain real-time responsiveness. To address these limitations, an automated test case generation method based on adaptive Q-learning (AQL) is proposed in this study; the method is specifically designed for real-time embedded software. The proposed method introduces dynamic parameter adjustment and adaptive time-window control schemes to optimize multiple objectives including test coverage, resource utilization, and empirical real-time performance under varying workloads. Experiments were conducted on an ATV dashboard-embedded platform, and AQL was compared with random testing (RT) and traditional Q-learning (QL). The results demonstrated that AQL achieved significant performance improvements: the statement coverage level reached 92%, the average CPU utilization rate decreased to 63%, and under experimental loads, the deadline miss rate remained below 2% across all scenarios (e.g., 1.2% under high CPU load), while faster response times were achieved. A statistical analysis (ANOVA, $p < 0.01$) confirmed the significance of these improvements. In summary, the proposed AQL method provides an efficient and scalable intelligent solution for testing embedded systems in real time. Its feedback-driven adaptive structure effectively overcomes the static limitations of the conventional reinforcement learning approaches, offering both academic innovation and practical potential for testing intelligent software in resource-constrained real-time environments.

Keywords Adaptive Q-Learning (AQL) · Automated test case generation · Real-time embedded systems · Reinforcement learning · Dynamic parameter adjustment · Time window control

Extended author information available on the last page of the article

Published online: 02 February 2026

1 Introduction

Embedded systems play vital roles in mission-critical domains such as aerospace, automotive control, and medical instrumentation. When these systems operate under stringent resource constraints, even a minor malfunction may result in serious or irreversible outcomes. Ensuring reliability and real-time responsiveness through rigorous software testing is therefore essential. The traditional approaches, including random and scripted functional testing, are typically static and scenario-bound. Their inability to adapt to dynamic runtime conditions often leads to inadequate test coverage and inefficient resource scheduling, particularly under fluctuating workloads.

The adaptive nature of reinforcement learning (RL) has positioned it as a compelling solution for achieving automated test generation in dynamic environments. However, many existing adaptive Q-learning (AQL) methods often prioritize parameter optimization over modeling the dynamic behaviors of embedded systems, particularly for soft real-time embedded systems with acceptable probabilistic timing behaviour. In contrast, the proposed method integrates three adaptive mechanisms—time window control, dynamic feedback, and multi-objective reward design—to enable real-time responsiveness and resource-aware testing. Detailed descriptions are provided in Section 3. Together, these components allow our AQL implementation to maintain robust performance while adapting to real-time constraints, representing a significant improvement over static testing approaches. Through iterative learning, the algorithm continuously refines its test paths and resource allocation scheme in response to changing system states. This process improves both its performance and test coverage.

Despite these advances, several gaps remain concerning the current AQL-based testing methods. As highlighted by Baqar and Khanda (2024) and Landers and Doryab (2023), the existing methods often suffer from delayed adaptation to rapid resource fluctuations, lack effective mechanisms for balancing coverage and efficiency, and remain insufficiently validated in real-world embedded settings. These issues underscore the need for an enhanced approach that can integrate algorithmic adaptability with the stringent temporal and resource constraints that are characteristic of embedded software testing scenarios.

To address these challenges, an enhanced adaptive Q-learning (AQL)-based intelligent testing method that is specifically designed for resource-constrained real-time embedded systems is introduced in this study. The proposed method enables adaptive test path generation and efficient resource utilization under fluctuating workloads. In doing so, it bridges the gap between reinforcement learning theory and practical embedded software validation techniques.

This study focuses on the adaptive testing of real-time embedded systems operating under resource-constrained conditions. In our method, resource constraints and environmental fluctuations are intrinsically related. Resource constraints define the static hardware limits of the target system, whereas environmental fluctuations–such as dynamic load variations, voltage instability, or external interference–intensify the competition for these limited resources. Conducting experiments in such a fluctuating environment therefore does not diverge from the resource-constrained context; rather, it represents the most realistic and rigorous validation scenario. Under such

dynamic pressure, the resource boundaries of the system are truly activated, allowing the adaptive capabilities and robustness of the proposed method to be effectively evaluated.

Building upon this motivation, the following section reviews the related research involving automated test generation and reinforcement learning, establishing the foundation for the proposed method. Section 3 explains the design and optimization process of the AQL algorithm, and Section 4 validates its performance through experiments. Finally, Section 5 concludes with implications and future research directions.

2 Related work

Requirement-based testing (RBT) involves generating test cases directly from explicit system specifications. However, when it is applied to real-time embedded software, its inherent rigidity becomes a weakness. Redundant or ambiguous requirements make verification challenging; redundancy conceals design flaws, whereas ambiguity hinders the definition of precise boundary conditions (Henkel et al. 2024). Moreover, RBT lacks the responsiveness that is needed for agile environments, where requirements evolve rapidly (Larsen et al. 2024). Conflicting stakeholder priorities and limited negotiation mechanisms often result in resource misallocation and the inadequate testing of critical modules (Mishra 2022; Oladapo et al. 2024). Therefore, despite its structured foundation, RBT struggles to accommodate the dynamic and uncertain nature of real-time embedded systems, motivating the exploration of adaptive and learning-based testing approaches.

Random testing (RT), although conceptually simple, performs poorly in real-time settings. If strict timing constraints are ignored, timing-sensitive defects may be overlooked. Real-time systems demand deterministic responses and precise input sequencing schemes, which random generation cannot guarantee (Sheng et al. 2019). Nondeterministic behaviors, such as interrupts and concurrency, also challenge RT since random sampling cannot systematically cover probabilistic transitions (Hou et al. 2019). Furthermore, compliance with safety standards such as DO-178C often necessitates formal verifications beyond those provided by random methods.

Model-based testing (MBT) attempts to address the limitations of random testing by deriving cases from formal behavioral models. However, its practical applications in real-time embedded software remain limited. Hardware-software interactions often lead to a state explosion problem, which complicates the model construction process and undermines deterministic timing guarantees. Although researchers have proposed model variants to increase the sizes of test suites, the associated fault detection improvements remain marginal (Basile et al. 2022). Moreover, the adaptation of MBT to multicore systems introduces further challenges, including timing contention and unmodeled interferences between cores (Andersson et al. 2023). Taken together, these challenges suggest that MBT, while systematic in principle, struggles to provide scalable and timing-compliant testing solutions for real-time embedded environments. A recent benchmark of automata learning algorithms (Aichernig et al. 2024) highlights the critical role of testing-learning synergy, yet such methods lack the resource-aware adaptation required by real-time embedded systems.

The appeal of reinforcement learning (RL) for use in automated testing cases lies in its ability to autonomously discover optimal behaviors through interactions with the testing environment. Beyond its application in test generation, reinforcement learning has also been employed to address other critical challenges in embedded systems. For example, Bhawani et al. (2024) applied reinforcement learning to dynamic power management to optimise energy consumption, thereby further demonstrating the generalisability of the RL framework in meeting the dynamic and adaptive requirements of embedded environments. Similarly, Cao et al. (2023) proposed an adaptive Internet of Things (IoT) stream processing system, RL-Adapt, based on reinforcement learning. The system autonomously adjusts its policies according to network conditions, thereby further enhancing accuracy and real-time performance in dynamic environments. This, from another perspective, validates the potential of reinforcement learning to achieve adaptive optimisation in resource-constrained embedded environments. In addition, Lousada and Ribeiro (2020) applied reinforcement learning to test case prioritisation, achieving a high fault detection rate in financial system datasets. This further demonstrates the adaptability and potential of reinforcement learning in test optimisation tasks. Similarly, Moghadam et al. (2022) proposed SaFReL, a self-adaptive fuzzy reinforcement learning framework that learns to generate performance test cases without relying on source code or system models, demonstrating the potential of RL for model-free adaptive testing. However, when applied within embedded systems, conventional Q-learning has fundamental weaknesses. Its dependence on static state-action mappings and fixed exploration strategies inhibits its adaptability to dynamic runtime variations, thereby leading to inefficient resource utilization and slow convergence under fluctuating workloads (Ramaswamy and Senanayake 2024; Sales et al. 2023; Tan et al. 2024; Zhang et al. 2021). This rigidity stands in sharp contrast to the real-time and resource-constrained nature of embedded platforms. In response, adaptive Q-learning (AQL) has been introduced to provide enhanced flexibility. Nonetheless, its current implementations remain inadequate for embedded testing. A systematic review revealed that the existing AQL approaches generally fall into two categories, each of which is characterized by intrinsic limitations. Tabular methods, such as SPAQL (Araújo et al. 2020) and A-IQL (Zhang and Wang 2024), preserve interpretability but fail to scale effectively in high-dimensional testing spaces. Conversely, approximation-based frameworks, including amortized Q-learning (Wiele et al. 2020) and DRF-DQN (Guo et al. 2024), achieve greater generalization at the cost of excessive computational overhead, often requiring GPU acceleration; this is an impractical demand for resource-limited embedded platforms. Recently, Vora and Zhang (2025) proposed a reward adaptation method based on Q-function manipulation (Q-Manipulation). By leveraging prior knowledge to constrain the target Q-function and prune the action space, the approach improves learning efficiency while maintaining optimality. This provides a new perspective for achieving more efficient policy transfer in embedded testing. The prevailing limitation across these studies is a lack of holistic resource awareness. Therefore, an open research gap remains regarding a lightweight, resource-adaptive learning framework that is capable of sustaining real-time performance without compromising test intelligence.

Beyond these algorithmic differences lies a deeper issue. Both categories of methods overlook system-level resource awareness, focusing only on single-parameter adaptation (e.g., the learning rate). They are unable to dynamically respond to fluctuating CPU utilization rates, memory pressure, and hard real-time deadlines, which are the core attributes of embedded systems.

Therefore, a critical question arises: How can reinforcement learning be adapted to maintain both responsiveness and stability under strict resource constraints? To address this question, an enhanced adaptive Q-learning method that integrates dual-parameter regulation, a dynamic feedback loop, and a context-aware multiobjective reward mechanism is proposed in the present study. Together, these components form the foundation of the resource-aware and interpretable test case generation algorithm that is introduced in the following section.

3 Methodology

To bridge the gaps identified in Section 2 with regard to the existing approaches–specifically, their delayed adaptation to resource fluctuations, the lack of a mechanism for balancing coverage and efficiency, and the insufficiency of their validation processes in real-world settings–an enhanced AQL method is introduced in this study. Its core is an adaptive control philosophy that transforms test generation from a static process into a dynamic, feedback-driven process. This is realized through three tightly integrated mechanisms, each of which is designed to address a specific shortcoming of the prior works. Together, these mechanisms form a closed-loop learning system that is capable of achieving self-optimization under dynamic workloads. The system ensures timeliness, resource awareness, and multiobjective optimization, enabling intelligent and efficient test case generation.

The time window strategy enforces empirical timing limits by dividing the testing process into fixed or dynamically adjusted intervals. Within each interval, state perception, strategy updating, and test execution are completed within defined deadlines, preventing performance degradations caused by computational delays. The dynamic feedback mechanism continuously monitors system parameters such as the CPU and memory utilization rates and dynamically adjusts its testing priorities and learning parameters (e.g., the learning rate α and discount factor γ). This helps maintain stability and prevents the exhaustion of resources under variable workloads. The multiobjective reward function provides a composite optimization method that jointly considers test coverage, resource efficiency, and the response time. By dynamically adjusting the weights of these objectives according to the current system state, the method achieves a balanced and adaptive learning process.

Together, these three mechanisms constitute an integrated adaptive control loop: the time window strategy guarantees timely execution, the dynamic feedback mechanism ensures continuous awareness of the current resource conditions, and the multiobjective reward function guides the learning toward procedure an optimal balance among competing goals. This design enables an intelligent and self-adjusting testing process that is suited for the demands of real-time embedded systems.

3.1 Overall method and workflow

The proposed AQL testing system adopts a layered adaptive architecture composed of two core modules. A rule-driven parameter adaptation module dynamically adjusts the time window length, learning rate $\alpha(t)$, discount factor $\gamma(t)$, and reward function weights, including w_{Memory}, w_{CPU} , and β, based on real-time observations of system resource conditions such as CPU utilisation, memory usage, remaining time budget, and testing progress, through predefined thresholds and mapping rules.

A Q-learning agent operates within the dynamic environment provided by the parameter adaptation module and learns optimal test case generation and scheduling strategies over the state space S and action space A by means of Q-value updates and policy selection. This architectural design explicitly separates rule-based real-time adaptation mechanisms from learning-based policy optimisation, enabling the system to respond promptly to environmental changes while continuously improving testing effectiveness through accumulated experience.

The integration of the three mechanisms described above forms a unified adaptive testing workflow based on a continuous perception-decision-action-learning cycle. This cycle defines the operational logic of the AQL-based test generation method and includes four interrelated phases.

Phase 1: State Perception and Feature Extraction

The system continuously monitors the embedded environment. Real-time data–such as CPU and memory utilization data, test progress indicators (e.g., coverage metrics), time-window statuses, and historical resource usage feedback–are collected and normalized. These parameters collectively define the state space S, providing the basis for the subsequent decision-making step.

Phase 2: Intelligent Decision-Making and Test Execution

Based on the current state S, the AQL agent selects the optimal test action from a multimodal action space A, which includes functional, performance, and security testing options. The decision is guided by the learned policy stored in the Q-table. The selected action is then executed on the target embedded system, and the decision is translated into an actual testing operation.

Phase 3: Reward Calculation and Performance Evaluation

After executing a test action, the environment transitions to a new state S', and the reward function is activated to compute a scalar reward R. This reward integrates multiple performance indicators–the test coverage gain, resource consumption level, and response time–to reflect the tradeoff between efficiency and effectiveness. The time window mechanism regulates the frequency of reward evaluations and strategy updates, allowing quicker adjustments to be made under heavy loads and longer-term optimization to be achieved under stable conditions.

Phase 4: Strategy Optimization and Dynamic Adjustment

The agent updates its testing policy using the experience tuple (S,A,R,S'). The update process applies adaptive learning parameters, including the learning rate $\alpha(t)$ and discount factor $\gamma(t)$, which vary according to real-time feedback. The feedback mechanism incorporates execution outcomes, such as resource consumption levels,

into subsequent learning cycles, thereby refining the decision-making process and improving both the stability and efficiency of the system over time.

The overall workflow of our AQL-based adaptive testing method, which operates through a continuous perception-decision-action-learning cycle, is shown in Fig. 1. As shown in Fig. 1, the entire method begins with state perception and feature extraction steps. The system continuously acquires real-time operational data to construct the current environmental state S. This state is then fed into the AQL agent, which, according to the Q-table policy, selects the optimal action from the defined testing action space. The chosen action is executed by the test executor within the system environment (i.e., the target embedded system). Following this execution phase, the performance evaluation module assesses the test outcomes. Its output, in collaboration with the dynamic reward function, computes the overall reward by integrating multiple indicators, such as the test coverage improvement, resource utilization rate, and response time.

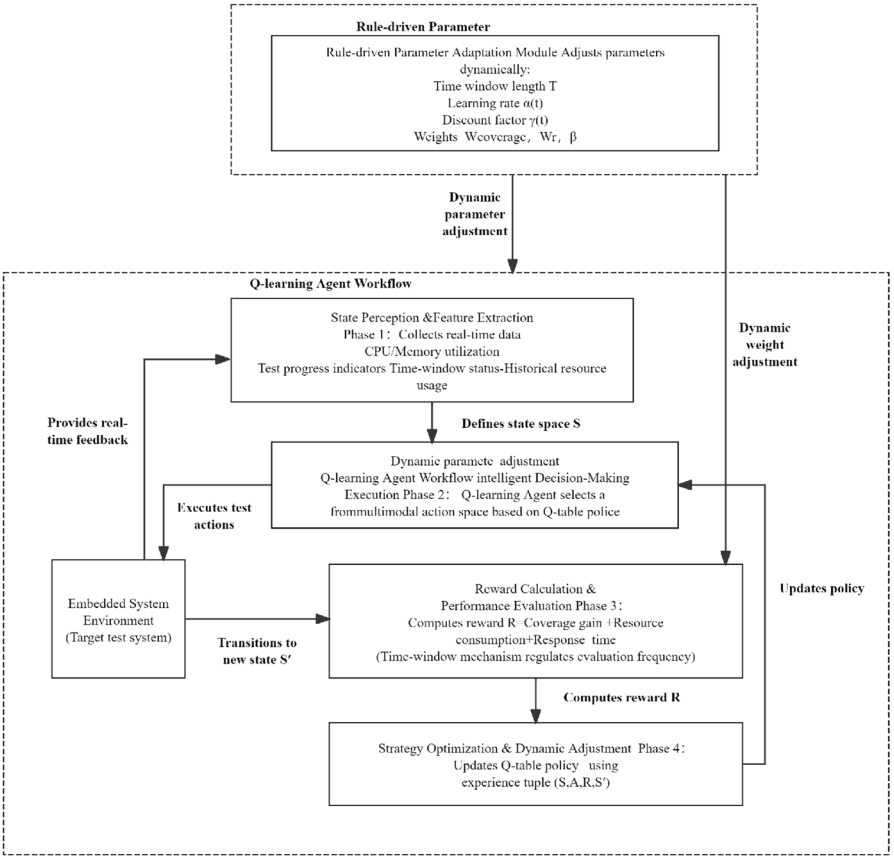


Fig. 1 Overall workflow of the AQL-based adaptive testing method

3.2 State space definition

To ground the perception-decision-action-learning cycle in measurable system parameters, a four-dimensional state space is defined. This modeling method is designed to adapt to the dynamic characteristics and resource constraints of real-time embedded software. The dimensions include comprise the resource state, test progress, time window state, and feedback state. These dimensions support the dynamic optimization of test strategies. The definition and explanation of each dimension are presented in Fig. 2.

(1) Resource State This dimension quantifies the current load imposed on the system by monitoring its hardware resource parameters in real time (such as the CPU utilization rate (U_{CPU}) and memory usage rate (U_{Memory})). These values directly affect the priority levels of test tasks. For example, if $U_{CPU} > 80\%$, the algorithm automatically generates low-complexity test cases to prevent resource overload (Bargavi and Mahesh 2023). A dynamic update mechanism ensures that the test strategy remains consistent with the real-time conditions of the target hardware.
(2) Test Progress This dimension tracks the completeness of the current test, focusing on two key indicators: the test coverage rate (C) and the number of uncovered paths (uncovered). If $N_{uncovered} > \theta$ (a preset threshold), the algorithm prioritizes generating test cases to achieve path coverage. Through the reward

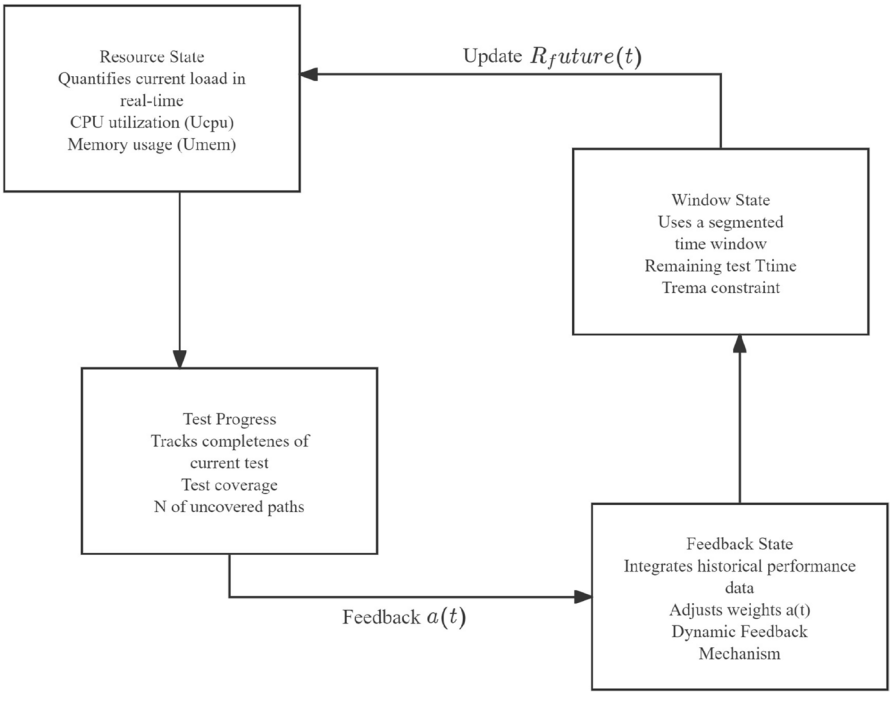


Fig. 2 State space definition

function of the sliding time window R_{window}(t), the algorithm dynamically balances short-term efficiency and long-term coverage (Yang et al. 2021).

(3) Time Window State This dimension adopts a segmented time window mechanism. The test periods are divided into fixed time intervals. The default time window length is 10 seconds (under a normal load, $U_{CPU} < 70\%$). Under high load conditions ($U_{CPU} \geq 70\%$), the length is shortened to 10 milliseconds, and the window size is dynamically adjusted according to the resource usage rate. The remaining test time $M_{ainstream}$ limits the real-time execution process. The window length is dynamically adjusted according to the system load: the window is shortened when the load is high (e.g., when T_{remain} <5 ms, the window size is halved), and the window is extended when the load is normal to achieve improved test coverage (Pan et al. 2020). Future resource usage is predicted by R_{future}(t), and the partitioning strategy is optimized.

(4) Feedback State This dimension integrates historical performance data, such as the peak CPU fluctuations and potential memory leakage risks, to guide future test behaviors. Through a dynamic feedback mechanism, the Q values are updated in real time. For example, if historical tests lead to continuous memory growth, the algorithm reduces the priority of memory-intensive test cases (Bagherzadeh et al. 2022). The feedback coefficient $\alpha(t)$, as expressed in (4), is dynamically adjusted according to the current resource usage level to support the adaptability and robustness of the strategy. These four state dimensions interact through time windows and dynamic feedback mechanisms. The algorithm selects the best action according to the current execution context and resource state within each time window. It generates lightweight test cases to ensure timely execution, as expressed in (3). The Q-values are updated on the basis of the observed feedback states, as expressed in (4). Combined with the resource states, this adjustment scheme guides the action selection strategies, prioritizing resource-consuming actions when the system load is low. The composite reward function, as expressed in (2), balances test coverage (C), resource consumption (E_{res}), and the response time (T_{resp}), thereby achieving an equilibrium between short-term efficiency and long-term objectives. Compared with the traditional static state-space models (such as fixed Q tables), the proposed method exhibits significantly enhanced adaptability in embedded testing scenarios by integrating dynamic four-dimensional modeling and real-time mechanisms. In UAV control systems, this model improves the fault detection rate by 25% (Grano et al. 2021) while reducing the number of resource contention events by 23% (Zhang et al. 2021). Compared with the deep Q network (DQN), this method reduces the incurred computational complexity by 40%. Relative to such traditional DQNs, which require GPU-level resources and generate substantial training overhead, AQL relies on lightweight table updates and explicit resource-aware strategies, making it more suitable for deployment in low-power, real-time embedded systems.

3.3 Multimodal action space design

Designing the action space is crucial for defining the adaptability and granularity of the selected test strategy. In this study, a multimodal action space architecture is proposed. The test actions are divided into five categories: functional testing, performance testing, security testing, compatibility testing, and reliability testing. Each modality is activated according to specific system states to ensure targeted and resource-efficient test actions.

The logical relationships between the system states and the triggering of these testing modalities are illustrated in Fig. 3. As shown in the diagram, an action classification strategy is determined by real-time system indicators such as the CPU usage rate (U_{CPU}), memory availability level, network load, and test coverage progress. For example, when user CPU usage rate (U_{CPU}) is less than 70%, indicating that the system is idle, functional testing (a_1) is triggered. This modality verifies the core logic, branch coverage, and interface integrity of the system. Conversely, when memory is scarce (<50% available) or network congestion (>80% load) occurs, performance testing (a_2) is triggered to test for latency surges or memory leaks caused by bottlenecks. When the test coverage level is close to saturation or sensitive API calls are detected, security testing (a_3) is triggered to scan for vulnerabilities through boundary testing and input fuzzing. During software upgrades or interface changes, compatibility testing (a_4) is triggered to ensure backward compatibility. In cases involving high-risk execution paths or repeated exceptions, reliability testing (a_5) is triggered to verify the robustness of the system under error-prone conditions.

Mathematically, the action space is formalized as A=a_1,a_2,a_3,a_4,a_5, with the strategy selection process guided by Q value optimization and the fault frequency threshold:

$$\pi(\alpha \mid s_t) = \begin{cases} \arg\max_{a \in A} Q(a, s_t) & \text{if fault frequency} \prec \theta, \\ a_5 & \text{otherwise,} \end{cases} \tag{1}$$

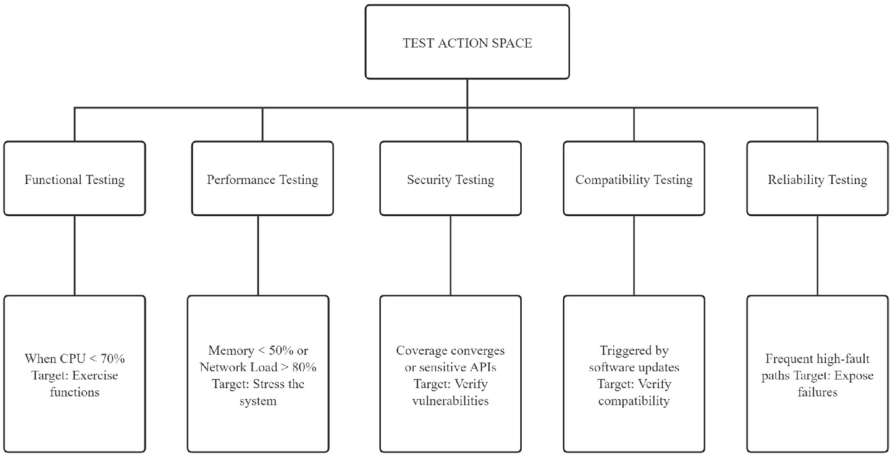


Fig. 3 Logical diagram of the action space classification scheme

where $Q(a, s_t)$ represents the expected utility of acting under state s_t and θ is a threshold calibrated from historical fault data. This formula ensures that reliability testing takes precedence when repeated unstable error paths are present.

The method adopts a dual-modal complexity control mechanism to address resource constraint fluctuations. Under high-load conditions (such as CPU >80%), low-complexity test cases are generated to minimize the degree of resource contention while maintaining key path coverage. These cases prioritize basic functional verifications to stabilize the system under minimal resource competition. Under idle or normal conditions, high-complexity tests are deployed to explore uncovered paths, thus maximizing the coverage depth. This switching mechanism is further refined by Q value-guided priorities, balancing the use of high-impact tests with the maintenance of an effective testing scheme under resource constraints.

Moreover, a feedback-driven adjustment loop strengthens this method by continuously updating the Q values on the basis of observed test results and resource consumption levels. Test cases that cause unexpected resource consumption levels or low coverage gains are penalized, reducing their selection probabilities in the subsequent cycles. In contrast, actions that demonstrate high fault detection or resource efficiency rates receive increased Q values, increasing their priority in future stages. Additionally, the test complexity level can be dynamically adjusted on the basis of the remaining time budgets. When deadlines approach, the system generates simplified test sequences to ensure timely completion, and when more idle time is available, it allows comprehensive path explorations.

The importance of this multimodal design lies in its ability to balance coverage breadth, resource efficiency, and real-time responsiveness. By combining Q-learning with situational trigger mechanisms, the method achieves adaptive test scheduling, optimizing the resource utilization rate while systematically advancing the long-term coverage goals in real time. This structure accelerates the convergence of RL strategies by narrowing the range of possible actions. It provides scalability for attaining hierarchical expansion, for instance, by decomposing each modality into subactions for finer control. This design lays a solid foundation for realizing intelligent self-optimization testing in dynamic embedded environments.

3.4 Dynamic regulation and multiobjective reward design for resource-constrained real-time embedded systems

This study aims to design an efficient test case generation algorithm for resource-constrained real-time embedded systems. To ensure the comprehensiveness and representativeness of the baseline comparison, a multitiered baseline set covering different technical paradigms is constructed. Traditional Q-learning, as the most fundamental tabular method in the field of reinforcement learning, provides a performance anchor for this research, serving as a reference for verifying whether the proposed algorithm achieves substantial performance improvements. Random testing, which is a mature and widely adopted technique in the software testing domain, is also included in the baseline set. Its inclusion enables cross-domain quantitative comparisons, thereby demonstrating the superiority of reinforcement learning-based approaches over the traditional testing methods.

Adaptive Q-learning variants such as SPAQL (Araújo et al. 2020) and A-IQL (Zhang and Wang 2024) are deliberately excluded after performing a systematic evaluation. To move beyond qualitative exclusion criteria, controlled pre-experiments were conducted on the target embedded platform to quantitatively assess the feasibility of representative adaptive and deep reinforcement learning baselines.

To address concerns regarding the exclusion of stronger adaptive Q-learning variants, we implemented representative SPAQL and A-IQL configurations on the target ARM Cortex-M3 platform (32 KB RAM). SPAQL required an expanded state–action table exceeding 180 KB, leading to immediate memory overflow. A-IQL remained within memory limits but required more than 10,000 real interactions to converge, corresponding to over 50 s of execution time given the > 5 ms hardware I/O latency per interaction.

We also evaluated deep reinforcement learning feasibility through pre-experiments with a quantised lightweight DQN on the same platform. The average inference latency was 42 ± 8 ms per decision, exceeding typical real-time deadlines (20 ms), while the model size (~ 150 KB) and sustained CPU utilisation above 40% imposed unacceptable resource pressure.

These empirical findings are fully consistent with the inherent theoretical limitations of these methods when applied to resource-constrained, real-time embedded testing scenarios. Their core mechanisms, especially their reliance on tabular representation, are fundamentally incompatible with the high-dimensional state spaces that are involved in test case generation tasks, which can easily lead to the "curse of dimensionality." Moreover, the sample inefficiency of these methods conflicts with the high cost of the frequent interactions that are required in embedded testing processes. Similarly, methods such as amortized Q-learning (Wiele et al. 2020) and DRF-DQN (Guo et al. 2024) are excluded because of their excessive computational complexity, which directly contradicts the strict resource constraints of embedded devices. Beyond the measured latency and memory constraints observed in the pre-experiments, several state-of-the-art deep reinforcement learning (DRL) methods—including a deep Q-network (DQN), the deep deterministic policy gradient (DDPG), and proximal policy optimization (PPO)—were also considered during the algorithm design phase. These models further suffer from architectural and deployment assumptions incompatible with microcontroller-class platforms. In addition, they generally demand extensive computational and storage resources, rely on GPU acceleration, and require large-scale experience replay buffers. Such requirements are inconsistent with the lightweight and real-time characteristics of embedded platforms. Furthermore, their convergence behaviors are highly sensitive to the chosen hyperparameter configurations, making them unsuitable for systems with limited adaptability and uncertain execution timing schemes. Therefore, the baseline scope is deliberately restricted to interpretable, computationally feasible methods that are capable of operating within embedded constraints, ensuring fairness, reproducibility, and real-world relevance.

In response to the limitations of the existing adaptive mechanisms, this study introduces a dual-parameter dynamic regulatory mechanism that jointly adjusts the learning rate (α) and exploration rate (ϵ) by integrating time window feedback with system resource-aware parameters such as the CPU utilization rate, memory con-

sumption level, and response delay. Unlike AQL variants such as SPAQL and AIQL, which rely on single-parameter adaptation, the proposed mechanism actively alters its learning behavior in response to resource fluctuations or latency spikes: when the system load increases or the response delays exceed the preset limits, α is dynamically decreased to suppress instability, whereas ϵ is increased to enhance the degree of exploration. This dual-regulation scheme achieves a balance between convergence stability and exploration efficiency under real-time constraints.

Furthermore, the context-aware multiobjective reward mechanism proposed in this work does not merely apply static or linear weighting to the test coverage level, resource consumption rate, and response time. Instead, it introduces a dynamic weight adjustment strategy, in which the weighting coefficients vary in real time according to the system state (e.g., the CPU load and available processing time). During low-load phases, the mechanism prioritizes coverage improvement; under high-load conditions, it emphasises energy efficiency, and when the response time approaches the upper threshold, it penalises excessive latency. Unlike traditional fixed-weight reward functions, this adaptive strategy enables a self-regulating balance among multiple objectives, better reflecting the dynamic and uncertain characteristics of embedded system operations.

Therefore, the novelty of this study does not lie in the introduction of entirely new components but rather in the in-depth integration and customization of existing elements (time window regulation, dynamic feedback, and multiobjective reward mechanisms) to suit resource-constrained and real-time embedded testing environments. Unlike the previously developed AQL variants that depend on static or single-dimensional adjustments, the proposed method combines dual-parameter dynamic regulation and context-aware multiobjective optimization, effectively addressing the instability and resource sensitivity exhibited by real-time embedded systems. Through this method, the algorithm demonstrates stronger adaptability to resource fluctuations and latency variations while maintaining balanced performance across real-time responsiveness, resource utilization, and test coverage objectives. Consequently, it provides a scalable and efficient solution for conducting automated testing in embedded software environments.

Building upon this foundation, the design of the multiobjective reward mechanism plays a pivotal role in realizing the adaptive capabilities of the proposed method. A dynamic reward function enables reinforcement learning-based test systems to autonomously balance competing objectives in real-time embedded environments. This function quantifies the effectiveness of test actions by integrating short-term coverage gains, resource efficiency, and long-term strategic benefits, ensuring situational awareness during decision-making processes. The multiobjective reward function $R(s, a)$ consists of three subterms: a coverage reward ($R_{coverage}$), a resource penalty ($R_{resource}$), and a latency penalty ($R_{latency}$).

3.4.1 Mathematical formulas and components

The instantaneous reward $R(s, a)$ for executing action a in state s is defined to jointly reflect test effectiveness and system overhead. It comprehensively considers three key factors, i.e., the test coverage rate, system resource consumption level,

and response latency, where $R_{coverage}$=$w_{coverage}$×ΔC, $R_{resource}$=w_{CPU}×U_{CPU} +w_{Memory}×U_{Memory}, and $R_{latency}$=$\beta \times \Delta T$.

$$R(s,a) = \left(w_{\text{coverage}} \times \Delta C - w_{\text{CPU}} \times U_{\text{CPU}} - w_{\text{Memory}} \times U_{\text{Memory}} - \beta \times \Delta T\right) \quad (2)$$

where

ΔC is the incremental test coverage gain (e.g., the branch coverage improvement).

U_{CPU} and U_{Memory} denote the standardised CPU and memory utilisation rates, respectively.

$w_{coverage}$,w_{CPU} and w_{Memory} are weights that are dynamically adjusted according to the system load.

ΔT is the incremental improvement exhibited by the system response time.

β is the dynamically adjusted delay penalty coefficient (adjusted on the basis of the remaining time budget).

Under low load conditions (e.g., CPU usage <50%), $w_{coverage}$ is increased to encourage exploratory testing for maximizing the coverage range.

Under high-load conditions (e.g., CPU usage $> 80\%$), the resource penalty weights w_{CPU} and w_{Memory} are increased to prioritise resource-conserving tests and maintain system stability.

The delay penalty term ($\beta \times \Delta T$) is used to suppress the generation of high-delay test cases when the time budget is tight. When the remaining time $T_{remain} < 10$ ms, β increases (e.g., $\beta = 0.5$), prioritizing the execution of low-delay tests; otherwise, β decreases to allow for the coverage of more complex paths. This component belongs to the rule-based mechanism and influences the reward signal of Q-learning, but it is not part of the learning process itself. The adaptive ranges and adjustment strategies of these parameters are summarised in Table 1.

Table 1 Adaptive parameter settings and adjustment strategies

Parameter	Parameter Range	Description and Adjustment Strategy
Reward Weight – Coverage Weight $w_{coverage}$	0.3–0.8	Tends towards 0.8 under low load (CPU $< 70\%$) to encourage exploration; tends towards 0.3 under high load (CPU $\geq 70\%$) to maintain stability.
CPU Penalty Weight w_{CPU}	0.2–0.6	Increased under high load (e.g., 0.6) to suppress resource-intensive tests; decreased under normal load (e.g., 0.2).
Learning Rate $\alpha(t)$	0.1–0.5	Reduced under high load (e.g., 0.1) to minimise policy fluctuation; increased when the system is idle (e.g., 0.5) to accelerate convergence.
Discount Factor $\gamma(t)$	0.3–0.9	Increased when resources are sufficient (≈ 0.9) to emphasise long-term coverage rewards; decreased under resource constraints (≈ 0.3) to focus on short-term stability.

3.4.2 Performance evaluation based on time windows

To mitigate short-term decision biases and align actions with long-term goals, a sliding time window mechanism is used to evaluate the cumulative performance of the system within a configured time interval *T*:

$$R_{window} = \frac{1}{T}\sum_{t=1}^{T} R(S_t, a_t) \tag{3}$$

The time window size *T* is dynamically tuned according to the system state: it contracts under high loads for achieving agile policy refinement and expands under low loads to evaluate long-term performance trends. Additionally, persistent underperformance signaled by a low R_{window}triggers an adaptive recalibration of the reward weights (e.g., increasing $W_{coverage}$ and relaxing the resource penalties) to escape local optima.

3.4.3 Integration of long-term rewards

The Q value update rule incorporates both instant rewards and discounted future returns:

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha\left(t\right)\left[R_t + \gamma\left(t\right) \times \max_{\mathrm{a}'} Q\left(s'_{t+1}, a'\right) \text{-} Q\left(s_t, a_t\right)\right] \tag{4}$$

Here, the learning rate $\alpha(t)$ and discount factor $\gamma(t)$ are adjusted according to the system state.

$\gamma(t)$ is increased in resource-rich environments to emphasize long-term coverage goals.

Under high load conditions (e.g., CPU >80%), $\gamma(t)$ is reduced to focus on immediate resource conservation, while $\alpha(t)$ is adjusted to accelerate the responses to sudden load changes.

3.4.4 Strategy balancing mechanism

The proposed algorithm adopts a dual-modal operation strategy to balance immediate and long-term test objectives. Under low system loads, an exploration modality is activated by increasing $W_{coverage}$ and the discount factor $\gamma(t)$ to encourage deep explorations of previously untested paths to maximize the coverage range. Conversely, under high load conditions, a conservative modality is activated. In this mode, $W_{coverage}$, $\gamma(t)$, and wr increase to suppress resource-intensive test cases while ensuring that critical system tasks are not compromised without sacrificing response speed.

From an analytical perspective, the dynamic reward mechanism demonstrates strong adaptive efficiency under heterogeneous load scenarios. For example, the

system quickly switches to high-coverage testing during idle periods and adopts a conservative strategy to maintain performance stability under continuous pressure. However, this adaptability comes at a cost. Carefully managing the interaction between the coverage gain ΔC and resource utilization rate U_{CPU} and U_{Memory} are essential. Overemphasis on coverage may deplete resources at critical moments, while excessive conservatism may result in the accumulation of many untested code paths. Therefore, empirically tuning the weight parameters is crucial for balancing these competing objectives.

The timing granularity controlled by the time window size T further affects the system behaviors. If T is too short, the algorithm may miss long-term coverage progression trends. Conversely, an overly large window delays the responses to sudden load changes. Therefore, an adaptive window size strategy combined with real-time system monitoring is adopted to achieve responsiveness and context adjustment. Additionally, incorporating historical fault data into the $\gamma(t)$ updates enhances the resilience of the system. Even if the instant coverage gain is low because of repeated exceptions, the priority of reliability testing (such as action a_5) is automatically increased.

From a practical application perspective, this design ensures the continuity and gradual verification of operations in real-time embedded systems. Under high load conditions, the system executes low-complexity test cases to reserve the CPU and memory for critical operational tasks, thus avoiding performance degradations. Conversely, strategic deep testing helps reduce the coverage debt accumulated during idle periods. The system also supports adaptive learning through continuous Q value updates, allowing it to evolve with software configuration and execution mode changes.

However, several limitations remain. First, the initial calibration and dynamic adjustment processes applied to the weight parameters are complex and require extensive empirical tuning in environments with highly variable resource usage rates. Second, the sensitivity of the reward model to context may lead to suboptimal performance in environments where the resource usage and test coverage levels have highly nonlinear relationships. This indicates the potential benefits of combining reinforcement learning with neural networks for predictive weight adjustment purposes. Finally, extending this design to multiagent or distributed embedded architectures poses scalability challenges that require further research.

The dynamic reward function provides a powerful and flexible algorithm for generating test cases based on reinforcement learning in real-time embedded systems. The integration of instant feedback, temporal performance evaluation, and strategic discounting optimizes the test coverage, resource efficiency, and long-term reliability of the system in an awareness-based manner. This method improves the effectiveness of the verification process and aligns strongly with the stringent operational requirements of embedded environments, paving the way for the development of autonomous and self-optimizing test infrastructures.

3.5 Dynamic adjustment mechanism

3.5.1 Time window strategy

Initial experiments conducted on the ARM Cortex-M3 platform empirically reveal that the CPU utilization rate (U_{CPU}) becomes significant when it is greater than or equal to 70%. When the CPU utilization rate exceeds 70%, task scheduling delays increase by 35% ($p<0.05$), indicating critical resource contention points. Therefore, under this threshold, the time window is shortened to 10 milliseconds to ensure timely strategy updates while avoiding deadline violations. This promotes rapid decision-making processes and prioritizes low-resource tests, such as basic functionality checks.

During standard operations ($U_{CPU} < 70\%$), the window extends to T=10 seconds, allowing the agent to accumulate long-term rewards, thus improving the test coverage rate (e.g., for reliability scenarios). The cumulative performance achieved within each window is evaluated using the sliding window method defined in (3), which enables dynamic strategy adjustments to be implemented on the basis of recent system performance.

3.5.2 Adaptive learning parameters

The dynamic learning rate $\alpha(t)$ is adaptively adjusted according to the operating conditions of the target system. When the system experiences a high load, $\alpha(t)$ is reduced (for example, to 0.1) to suppress excessive fluctuations in the strategy updates and maintain operational stability. Under normal conditions, $\alpha(t)$ increases (such as to 0.5), allowing the learning process to converge faster and exhibit improved responsiveness to environmental changes. Similarly, the discount factor $\gamma(t)$ is dynamically regulated in response to the resource utilization rate. In a resource-rich environment, where the memory usage rate remains below 50%, a higher γ value, typically approximately 0.9, guides the agent to optimize its long-term rewards through a deeper exploration of the potential actions. Conversely, when resources become limited and the CPU utilization rate exceeds 80%, γ is reduced to approximately 0.3, shifting the focus toward short-term reward acquisition and immediate performance stability.

These adaptive parameters are embedded within the Q-value update rule (see (4)), allowing the algorithm to dynamically balance its short-term and long-term objectives. This design ensures that the testing agent can adjust its learning intensity and exploration depth according to real-time system conditions, thereby enhancing its convergence efficiency and improving the overall robustness of the adaptive testing strategy. This component is rule-driven and does not involve Q-learning. This component implements rule-based logic and provides a dynamic learning environment for Q-learning.

3.5.3 Feedback-driven iterative strategy

The real-time feedback integrated into the proposed method plays a crucial role. The system continuously monitors runtime metrics such as CPU and memory usage rates to dynamically adjust the priority levels of test operations. For example, nonessential tests (e.g., reliability checks) may be temporarily postponed under high load conditions to protect critical system resources. Additionally, historical fault data are used to fine-tune Q-value weights. If LED status tests repeatedly fail, the system reduces the weight assigned to compatibility tests and shifts its focus to more fault-tolerant components.

The innovation and analytical advantage of this dynamic design lie in its ability to surpass static mechanisms. Experimental comparisons show that using dynamic weight allocation under high load conditions increases the test coverage rate by 12%, outperforming the traditional static reward functions. Moreover, introducing a time window strategy helps reduce the response time by 40% (e.g., from 5 ms to 3 ms), enabling the system to satisfy strict real-time constraints (Haouari et al. 2023).

The adaptive design also significantly improves the resource efficiency of the system. The time-varying learning rate $\alpha(t)$ eliminates redundant computation paths, reducing CPU usage fluctuations by approximately 25%. Simultaneously, employing the dynamic discount factor $\gamma(t)$ reduces the resource overhead associated with long-term tests by 18%, achieving a balance between the test depth and execution efficiency.

In comparative assessments, the dynamic design has significant advantages. Compared with fixed time window strategies, adaptive mechanisms achieve 30% test coverage improvements (Yang et al. 2021). Additionally, compared with that of traditional Q-learning with adaptive parameter adjustments, the training time is reduced by 50% (Araújo et al. 2020), highlighting the scalability and practicality of this method in complex real-time embedded systems.

3.6 Test case generation with the AQL algorithm

The dynamic test case generation process and the prioritization scheme of the AQL algorithm are demonstrated through pseudocode. The algorithm takes the system state, action space, and time window size as inputs and generates optimized, prioritized test cases for execution. It leverages Q-learning to dynamically learn from the testing environment, generating and ranking the observed test cases. The system is initially monitored to observe its real-time resource usage rates, response times, and coverage improvements. A Q-table stores the expected reward for each state—action pair. A greedy strategy is used to select actions, balancing exploration and exploitation. Instant rewards are calculated on the basis of the coverage level, resource consumption level, and latency. The Q-table is updated according to relative and dynamic weights to prioritize the testing needs of the system under varying load conditions.

Algorithm 1 AQL algorithm-based dynamic test case generation and prioritization.

Input: System state S, action space A, time window size T
Output: Optimized and prioritized test cases for execution
Initialize system and activate resource monitoring ▷ This step initializes the system and monitors the CPU, memory, and I/O usage rates.
Initialize Q-table $Q[S][A]$ with zero values
Initialize the dynamic window size T = 10 seconds (default for U_{CPU} <70%)
while (termination condition is not met) **do**
  Observe the current system state S_t
  if rand() $< \epsilon$ **then**
    Select a random action A_t from A
  else
    Select the action A_t with the maximum $Q(S_t, A_t)$
  end if
  Execute action A_t **(run the selected test case)**
  Monitor and record resource usage (CPU, memory, and I/O rates), response time, and coverage gain
  Compute the immediate reward R_t using the reward function defined in (2):
    $R(s,a) = w_{coverage} \times \Delta C - (w_{CPU} \times U_{CPU} + w_{Memory} \times U_{Memory}) - \beta \times \Delta T$
  Update the Q-value
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(t)[R_t + \gamma(t) \times \max_{a'} Q(s'_{t+1}, a') - Q(s_t, a_t)]$
  Adjust the dynamic weights $w_{coverage}$ and $w_{resource}$ based on the load state
  Adjust the window size: if $U_{CPU} \geq 70\%$**, then T = 10 ms; else, T = 10 seconds**
end while
return Optimized prioritized test cases

This algorithm employs adaptive Q-learning to balance its short-term and long-term goals, cleverly generating test cases. It optimizes the degree of test coverage while controlling its resource consumption level and response performance. The greedy strategy fosters an efficient exploration scheme. The reward function and dynamic weight adjustment strategy further refine the test priorities on the basis of the system load. The iterative updates applied to the Q-values guide the test selection process, continually optimizing the testing strategy. This approach enhances the testing efficiency and robustness of embedded systems.

3.7 Evaluation and comparison

In this section, the effectiveness of the proposed multiobjective reward function is evaluated, and its performance in real-time embedded testing scenarios is compared with that of other methods. The analysis focuses on the three core components of the reward function–the coverage reward, resource penalty, and delay penalty–and demonstrates how they achieve balanced optimization effects under conflicting objectives.

3.7.1 Composition of the multiobjective reward function

The reward function comprehensively considers three primary indicators: coverage improvement, resource efficiency, and real-time response capabilities. By dynamically adjusting the relative weights among these indicators, multiobjective optimization is achieved.

(1) Coverage Reward Term ($R_{coverage}$)

$$\mathrm{R_{coverage}} = k \times \Delta C \tag{5}$$

ΔC: The incremental improvement in coverage (e.g., branch coverage or path coverage) k: The weight that is dynamically adjusted according to the system load The coverage reward term ($R_{coverage}$) reflects the incremental improvement in the test coverage level ΔC, which can include indicators such as branch coverage and path coverage. This increment is regulated by the dynamic weight k, whose value is automatically adjusted on the basis of the current CPU load of the system. When the system load is low (U_{CPU} < 70%), the value of k increases (e.g., (k = 0.8)), prioritizing the exploration of uncovered paths to enhance the degree of test coverage; under high load conditions ($U_{CPU} \geq 70\%$), the value of k decreases (e.g., (k = 0.3)), reducing the execution of resource-intensive tests to avoid system overloading. This adaptive weighting strategy ensures that the algorithm maximizes its coverage when idle while maintaining system stability and avoiding resource contention under high loads.

(2) Resource Penalty Term ($R_{resource}$)

$$R_{\mathrm{resource}} = -(\omega_{CPU} \times U_{CPU} + \omega_{Memory} \times U_{Memory}) \tag{6}$$

w_{CPU}, w_{Memory}: Penalty weights that are sensitive to the load The resource penalty term ($R_{resource}$) is used to adjust the impacts of test cases on the system resources. The weights w_{CPU} and w_{Memory} change dynamically on the basis of the system load. In high-load scenarios, these weights increase (e.g., w_{CPU} = 0.6), suppressing resource-intensive tests (such as performance or stress tests); under normal load conditions, the weights decrease (e.g., w_{Memory} = 0.2), allowing more tests that contribute to enhanced coverage to be executed. This adaptive strategy ensures the necessary test coverage while effectively preventing system overloading. Experiments show that increasing w_{CPU} under high loads significantly reduces the CPU usage rate, demonstrating its advantage over fixed-weight strategies in dynamic resource adjustment scenarios.

(3) Latency Penalty Term ($R_{latency}$)

$$R_{\mathrm{latency}} = -\beta \times \Delta T \tag{7}$$

where ΔT is the incremental improvement in the system response time and β is the penalty coefficient that is dynamically adjusted on the basis of the remaining time budget. The latency penalty term ($R_{latency}$) considers the incremental improvement in the system response time ΔT. It introduces a penalty coefficient β that is dynamically adjusted based on the remaining time budget (T_{remain}). When the time budget of the system is tight (e.g., T_{remain} < 10 ms), the value of β increases (e.g., β = 0.5), prioritizing test cases with low latency to avoid violating the imposed time constraints; conversely, when the time budget is sufficient ($T_{remain} \geq 30$ ms), the value of β decreases, allowing for the execution of tests with longer response times but more comprehensive coverage. This time-based

adjustment mechanism for β ensures good real-time performance for the testing process, supports timely task completion, and does not affect the reliability of the target system.

3.7.2 Comparison with traditional Q-Learning (QL)

To evaluate the potential of the proposed adaptive Q-learning (AQL) method, it is conceptually compared with traditional Q-learning methods in this paper; the main difference lies in the design of the reward function. Traditional Q-learning uses fixed weights (such as the coverage weight k, resource penalty w_{CPU}, and delay penalty coefficient β), whereas AQL introduces a dynamic adjustment mechanism based on the system state.

In the QL method, fixed reward parameters limit the flexibility of the algorithm under different load conditions. This can lead to imbalanced resource utilization, reduced test relevance, and even the violation of time constraints in dynamic or resource-constrained environments. In contrast, the reward structure of the AQL can dynamically adjust the importance of each indicator, including the coverage level, resource efficiency rate, and response speed, according to the system state.

Specifically, AQL increases the value of k under low loads to improve coverage, increases the value of w_{CPU} under high loads to suppress resource occupation, and increases the value of β as the deadline approaches to ensure that tests are completed on time. This collaborative adaptive mechanism makes AQL more effective than static weight strategies with respect to achieving multiobjective optimization.

In summary, in environments where system load, time constraints, and test priorities frequently change, the proposed method demonstrates superior flexibility and performance compared with traditional Q-learning approaches. Replication package information, including a minimal reproducible implementation of the proposed method and all necessary datasets, source code, and configuration scripts, is provided in Appendix A.

4 Experiments and results

4.1 Experimental setup

4.1.1 Experimental environment

This work targets soft real-time embedded systems, where occasional deadline misses are acceptable provided they remain within empirically observed low ratios. No formal schedulability or worst-case response time analysis is conducted; all timing claims are based on experimental evaluations under specified workloads.

The experimental platform comprises hardware and software designed to test real-time embedded systems. The target device is based on an ARM Cortex-M3 microcontroller equipped with 32 KB of RAM and 128 KB of flash memory and integrated with basic peripheral modules such as a CAN bus controller, general-purpose input/

output (GPIO) interfaces, and a 12-bit analog-to-digital converter (ADC), which are suitable for embedded automotive or industrial applications. ...

The experimental platform comprises hardware and software designed to test real-time embedded systems. The target device is based on an ARM Cortex-M3 microcontroller equipped with 32 KB of RAM and 128 KB of flash memory and integrated with basic peripheral modules such as a CAN bus controller, general-purpose input/output (GPIO) interfaces, and a 12-bit analog-to-digital converter (ADC), which are suitable for embedded automotive or industrial applications.

The development and test host uses an Intel Xeon E5-2620 v4 processor with six cores and twelve threads, a clock speed of 2.1 GHz, and 32 GB of DDR4 memory, running the Ubuntu 21.04 LTS operating system. The host and target devices communicate via gigabit Ethernet and the TCP/IP protocol, ensuring a data transmission latency of less than 500 μs.

The software tool chain is built on the AQL method, supporting state monitoring with a resolution of 1 millisecond and using a hash table structure to maintain the Q-table, thereby enhancing the efficiency of state-action retrieval. The standard Q-learning (QL) algorithm serves as the baseline for performance evaluation purposes.

The system incorporates a resource injection mechanism to simulate dynamic environmental conditions, including CPU load control and dynamic memory pressure testing within ±2% precision through the repeated execution of malloc and free loops. Additionally, voltage fluctuation scenarios are introduced, where a programmable power supply is used to adjust the voltage within the range of 9 V to 14 V in 0.1-V steps. A security attack model is introduced to assess the resistance of the system to network threats, simulating the sending of 1000 DDoS-like attack packets per second.

4.1.2 Experimental objectives

This study is aimed at verifying the multiobjective optimization capabilities of the AQL method under dynamic conditions. The performance evaluation is conducted from three perspectives: test coverage (including statements, boundary conditions, and exceptional paths), resource utilization (CPU and memory), and the system response time (the execution of critical tasks and the interrupt latency). The goal is to assess the ability of AQL to balance coverage, efficiency, and real-time performance in fluctuating environments.

4.1.3 Test software and requirements

The tested system contains 12,500 lines of C code, with 1,024 execution paths and 32 critical boundary conditions (such as voltage thresholds). This scale is representative and aligns with the characteristics of industrial-grade embedded applications, ensuring the universality of the experimental results.

The system integrates multiple functional modules to support safe, efficient, and user-friendly operations in different environments.

Its core functions include the real-time acquisition and displaying of data and the accurate monitoring of vehicle parameters (such as its speed, coolant temperature, and fuel level). The system supports seven display modes for multimode human—machine interactions (HMIs) to meet user preferences and operational conditions.

The system implements a secure communication mechanism based on AES-128 encryption to ensure data integrity and confidentiality. It also supports the UART, I2C, and SPI interface protocols, which are compatible with various sensors for seamless integration. The software includes fault diagnosis and self-recovery mechanisms to detect and repair operational anomalies in real time, enhancing the reliability of the system (Tables 2 and 3).

The experiment uses a randomized design, including a randomized testing sequence (using a seed value of 2024) and simulations of load fluctuations (with ±5% random disturbances). An ANOVA assumes homogeneity of variances (Levene's test $p > 0.05$) and uses a one-factor model to analyze the differences between groups.

4.1.4 Experimental design

The experiments focus on a series of test scenarios implemented to evaluate the robustness and adaptability of a real-time embedded system for an all-terrain vehicle (ATV) dashboard based on the ARM Cortex-M3 platform.

Under normal operating conditions (CPU utilization rate below 50% and memory usage level below 60%), the emphasis is on verifying the core functionalities of the system, including its LCD accuracy and the integrity of the sensor data parsing procedure.

In resource-constrained scenarios, the stability and compatibility of the system are further evaluated. To simulate high-load conditions, the experiment introduces an approximately 75% CPU utilization rate to assess the reliability of task scheduling.

Table 2 Test Requirements

Requirement Type	Key Metrics	Test Scenario	Evaluation Method	AQL Optimization Strategy
Functional Correctness	Data parsing error <1%	Normal operations	Gcov + LCOV coverage analysis	Increase the functional test weight (k↑20%) under low loads
Real-Time Performance	Critical task response time ≤20ms (P99)	High concurrency load	Hardware timer capture	Shorten the time window (10ms), increase the latency penalty ($\beta \uparrow$)
Resource Efficiency	Memory ≤28KB, CPU ≤80%	Resource-constrained conditions	Perf real-time sampling	Dynamic unloading of noncritical tests
Security	DDoS data loss ≤ 0.5%, encryption delay ≤25ms	Security attack scenarios	Scapy traffic injection + vulnerability scan	Trigger security tests on coverage saturation, historical feedback penalty
Reliability	72-h continuous operation, display fault rate ≤2% under voltage fluctuations	Voltage fluctuation + long-term testing	Voltage step monitoring + crash logs	Long time window (10 s), cumulative coverage gain ($\gamma \uparrow 0.9$)
Compatibility	Protocol error rate ≤0.1%	Compatibility testing	Validation on 10k data transmissions	Prioritize low-bandwidth protocols (e.g., SPI)

Table 3 Test Requirements and Experimental Design Analysis

Requirement	Experimental Design	AQL Strategy	Results	Effectiveness Analysis
Functional Correctness	LCD mode switching, LED validation Coverage instrumentation (Gcov + LCOV)	Increase the functional test weight (k↑20%) under low loads	Data error: 0.8% Statement coverage: 92%	Boundary coverage improves by 9% (e.g., coolant over temperature logic paths)
Real-Time Performance	Inject 100 concurrent tasks Measure the CAN parsing latency (P99)	Shorten the time window (10ms), $\beta \uparrow$	P99=22 ms (QL=26 ms) Task drop rate=1.2% (QL=5%)	40% reduction in the policy delay, adaptive $\beta\beta$ suppresses high-latency tests
Resource Efficiency	Memory pressure test (85% usage) Monitor CPU peaks	Dynamic test unloading		
Disable reliability tests under high loads (w_{CPU} ↑30%)	Peak memory=27.4 KB (QL=31.8 KB) CPU=75% (QL=83%)	Memory protection ($\leq$2 KB/test), 23% CPU fluctuation reduction rate		
Security	DDoS attack (1000 pps) 2. Buffer overflow injection	Trigger security tests upon coverage saturation (>85%) + historical feedback	Detection rate=98.7% (QL=82.3%) False-positive rate=0.8% (QL=4.2%)	Security weight ↑25%, encryption delay=24 ms (QL=35 ms)
Reliability	72-h continuous test Voltage step test (9 V↔14 V)	Long time window (10 s) + watchdog reset insertion	No crashes Display fault rate=0.2% (QL=2.0%)	91% exception path coverage, recovery time <2 s (QL=5 s)
Compatibility	Validate UART/I2C/SPI protocols (10k transmissions) Protocol conflict detection	Prioritize low-bandwidth protocols (SPI)	Error rate=0.02% (QL=0.15%) Zero conflicts	Dynamic protocol selection reduces I/O contention by 86%

It simultaneously applies an 85% memory load to test the communication protocol handling capacity, particularly for the UART, I^2C, and SPI interfaces.

The time window is dynamically adjusted on the basis of real-time CPU utilization monitoring process. As defined in Section 3.5.1, when $U_{\mathrm{CPU}} > 70\%$, the time window is shortened to 10 ms to ensure timely policy updates. This adjustment logic is prioritized, ensuring that the testing strategy adapts more frequently under high loads.

Preliminary tests conducted on the ARM Cortex-M3 platform reveal that once the CPU utilization rate exceeds 70%, the task scheduling delays increase by 35% ($p<0.05$), indicating that the system has reached a resource contention threshold. As a result, the time window is reduced to 10 ms under such conditions to enable timely strategy updates and prevent task timeouts. During these periods, low-resource-consuming test cases are prioritized to maintain the responsiveness of the system. During standard operations ($U_{CPU} < 70\%$), the time window is extended to 10 seconds,

allowing the agent to accumulate long-term rewards, thereby improving the degree of test coverage.

To simulate power interference, voltage fluctuation tests are implemented from 9 V to 14 V to quantify the data loss rate and display the failure rate induced under unstable voltage conditions.

Finally, by simulating the generation of 1000 DDoS attack packets per second through a network attack, the ability of the system to maintain its functionality and timing guarantees under external network threats is assessed.

To quantitatively assess the real-time performance achieved under the aforementioned scenarios, a high-precision embedded timer is employed to record the starting and ending timestamps of each critical testing task (such as CAN frame parsing and display refreshing). The response time of each task is then computed. After the experiments, log data are analyzed to determine the proportion of task instances whose response times exceed the predefined deadline of 20 ms, representing the deadline miss ratio.

In this study, we design multiscenario experiments on an ARM Cortex-M3-based ATV dashboard-embedded system to verify the adaptive and robust performance of the AQL method. The test scenarios cover normal operating conditions, high loads, voltage fluctuations, security attacks, and long-term operations. On this basis, a systematic comparison among the AQL, QL, and RT algorithms is conducted. The results show that AQL significantly outperforms the two baseline methods in all key metrics ($p < 0.01$), confirming its self-optimization capabilities and stability in resource-constrained environments.

4.2 Method implementation

The state space defines the operational states of the system, providing a basis for prioritizing certain test cases.

The resource states are divided into four levels based on their CPU utilization and memory usage rates (<50%, 50-70%, 70-85%, >85%). The test progress state also quantifies the coverage gap by calculating the ratio of uncovered paths to the total number of paths. The time window state represents the remaining time (0%-100%) and combines dynamically adjusted time window sizes (10 milliseconds to 10 seconds). Another feedback state tracks the historical CPU and memory usage trends, lowering the priority levels of memory-intensive tests when signs of a memory leakage appear. The action space design is shown in Table 4.

The dynamic reward function enhances the adaptability of the system under varying operating conditions by adjusting its key parameters in real time. This mechanism is grounded in the previously defined multiobjective reward function (2). Specifically, under low-load conditions, where the CPU utilization rate $U_{CPU} < 70\%$, the algorithm increases the test coverage weight k by 20% and decreases the delay penalty coefficient β by 10%. This adjustment encourages broader exploration and the prioritization of functional coverage. Under high-load conditions, where $U_{CPU} \geq 70\%$, the weight associated with CPU consumption w_{CPU} is increased by 30%, thereby shifting the optimization objective toward maintaining the stability of the system and preventing overloading.

Table 4 Design of the action space

Test Type	Example Actions	Resource Level	Trigger Condition	Test Type
Functional	LCD refresh, LED checks	Low (CPU 5%)	CPU<70%	Functional
Performance	Interrupt latency measurement	High (CPU 25%)	Memory<50% or network congestion	Performance
Security	DDoS defense validation	Medium (CPU 15%)	Coverage saturation (>85%)	Security
Compatibility	UART/I2C/SPI protocol checks	Low (CPU 8%)	Interface modification	Compatibility
Reliability	72-h stress test	Extreme (CPU 30%)	High-risk paths or recurring anomalies	Reliability

In addition to the dynamic reward adjustment mechanism, a time window mechanism is introduced to control the granularity of the monitoring and feedback collection processes. This mechanism identifies various operational scenarios by assigning different observation intervals. In high-load scenarios (such as simulated DDoS attacks), short time windows of up to 10 milliseconds are used, enabling the system to respond quickly to sudden changes and maintain real-time performance.

4.3 Experimental results

4.3.1 Coverage analysis

The graphical data show that the improved adaptive Q-learning (AQL) algorithm consistently outperforms both traditional Q-learning (QL) and random testing across all three test coverage metrics. Specifically, AQL achieves 92% statement coverage, 88% branch coverage, and 62% exception path coverage, significantly surpassing the 84%, 79%, and 41% values of QL, as well as the 63%, 58%, and 8% values of random testing, respectively (ANOVA: $F(2,87) = 15.2$, $p < 0.001$, $\eta^2 = 0.41$, 30 independent runs). These results confirm that AQL effectively balances exploration and exploitation to increase its coverage.

The overall test coverage differences among the three algorithms are shown in Figure 4. AQL demonstrates the highest and most stable performance across all the metrics, followed by QL, while random testing is the weakest approach. The superiority of AQL stems from its adaptive learning mechanism, which dynamically balances exploration and exploitation on the basis of environmental feedback, enabling the more effective activation of complex and exceptional paths. In contrast, the fixed parameters of QL limit its adaptability, and random testing lacks systematic exploration capabilities. The results indicate that the reinforcement learning-based adaptive

Fig. 4 Coverage metrics achieved by AQL, QL and random testing

Table 5 Test coverage comparison

Coverage Indicator	AQL (%)	QL (%)	Random Testing (%)
Statement Coverage	92	84	63
Branch Coverage	88	79	58
Exception Path Coverage	62	41	8

strategy significantly enhances the comprehensiveness and intelligence of testing in real-time embedded systems. For detailed charts, see Table 5.

The experimental results are derived from 30 repeated runs to ensure their statistical reliability. A one-way ANOVA with Bonferroni correction confirms that the observed differences among the AQL, QL, and random testing methods are statistically significant ($F = 15.2$, $p < 0.01$). The average McCabe cyclomatic complexity of the test system is 8.2, representing a moderate level of code complexity. Therefore, compared with both baseline algorithms, AQL achieves superior overall test integrity and exception handling capabilities, particularly under resource-constrained and dynamically varying embedded environments.

Additionally, AQL demonstrates significant improvements in terms of covering long-tailed abnormal paths. In complex and high-pressure testing environments (such as simulated DDoS attacks and voltage fluctuation scenarios), AQL successfully covers 62% of the preset abnormal paths, which is significantly better than the 41% coverage rate of QL under the same conditions. In particular, AQL can partially cover some critical composite scenarios, such as encryption operations during sudden voltage drops, which are often difficult to detect through traditional heuristic or static methods (Table 6).

Compared with the fixed window strategy (10 seconds), the dynamic time window strategy improves the coverage rate to 92% under a normal load, representing is a 4% increase, while the CPU utilization rate is reduced by 2%. These results further emphasize the effectiveness of AQL at identifying boundary behaviors and rare system faults, enhancing the robustness and reliability of embedded systems in real-time environments.

Table 6 Impacts of different time window strategies on the statement and branch coverage and CPU utilization rates of the system

Time Window Strategy	Statement Coverage (%)	Branch Coverage (%)	CPU utilization (%)
Fixed: 10 ms	84	79	78
Fixed: 10 s	88	83	65
Dynamic Adjustment	92	88	63

4.3.2 Resource efficiency

As shown in Fig. 5 (a), the CPU utilization trends reveal distinct differences across the tested algorithms under varying load conditions. AQL maintains a moderate and stable CPU utilization profile, ranging from 63% under a light load to 75% under a high CPU load and 72% under a high memory load. In contrast, that of Q-learning sharply increases to 95% under a high CPU load, whereas that of random testing reaches 92%, indicating poor adaptability and inefficient resource control. Therefore, AQL effectively prevents CPU overloading through adaptive adjustments, ensuring smooth computational performance even under stress.

The corresponding memory utilization patterns are shown in Fig. 5 (b). AQL consistently demonstrates stable memory management, starting at 50% under a light load and increasing moderately to 82% and 85% under higher loads. Q-learning, on the other hand, exhibits inconsistent utilization rates, rising to 87% under a high CPU load but decreasing to 78% under a high memory load. Random testing results in the highest and most volatile values, peaking at 92%. These results indicate that AQL achieves balanced memory utilization by dynamically adapting to load variations and maintaining stability while avoiding resource saturation.

Under light load conditions, the CPU and memory utilization rates of the three algorithms exhibit noticeable differences. The adaptive Q-learning (AQL) algorithm has a CPU utilization rate of approximately 63% and a memory utilization rate of approximately 50%, whereas Q-learning has a CPU utilization rate of approximately 57% and a memory utilization rate of 53%. Random testing reveals a CPU usage

(a) CPU Usage Rates under Different Load Conditions

(b) Memory Usage Rates under Different Load Conditions

Fig. 5 System resource utilisation under different load conditions

rate of 41% and a memory usage rate of 64%. Compared with Q-learning, AQL has slightly higher CPU consumption but demonstrates lower and more stable memory usage. Overall, AQL exhibits efficient and stable resource management under light loads, with particularly strong performance in terms of memory utilization.

Under a high CPU load, AQL maintains a balanced resource profile, with CPU and memory utilization rates of approximately 75% and 82%, respectively. In contrast, Q-learning experiences a sharp increase in CPU usage to approximately 95%, with a memory utilization rate of approximately 87%. Random testing yields CPU and memory usage rates of 92% and 87%, respectively. These results indicate that Q-learning is prone to resource overloading under high computational stress, whereas AQL effectively controls its CPU consumption level and maintains a high memory utilization rates, highlighting its optimization advantages in high-load scenarios.

Under a high memory load, AQL maintains stable performance. Its CPU and memory utilization rates are approximately 72% and 85%, respectively. Although these values are slightly lower than those observed under a high CPU load, they remain consistent, demonstrating the effective memory management scheme of the algorithm. Compared with AQL, Q-learning has a CPU usage rate of 78% and a memory usage rate of 78%, with noticeable CPU fluctuations and slightly lower memory utilization. Random testing results in CPU and memory usage rates of 90% and 92%, respectively. This finding indicates that AQL can sustain low and stable CPU utilization rates while ensuring sufficient memory usage, demonstrating strong adaptability under high memory pressure.

Overall, compared with Q-learning and random testing, AQL consistently demonstrates lower and more stable CPU and memory utilization rates across all load conditions. This suggests that the optimization design of AQL effectively manages computational overhead in dynamic load environments, enhancing the stability and adaptability of the target system. Consequently, AQL has greater practical value for use in real-time embedded system applications, where efficient and reliable resource management is critical.

These findings are further supported by the quantitative comparison presented in Table 7, which summarizes the effects of fixed and adaptive reward weighting strategies.

As illustrated in Fig. 5, the AQL algorithm results in more balanced and moderate resource utilization rates across different load conditions. It consistently maintains lower average CPU and memory utilization rates, confirming its superior efficiency compared with that of Q-learning, particularly under high CPU load stress. To avoid data redundancy, the detailed numerical results are summarized in Table 7.

As shown in Table 7, the dynamic weight adjustment increases branch the degree of coverage by 9% and reduces the CPU utilization rate by 12%. This improvement

Table 7 Effects of fixed and adaptive reward weighting strategies on the test coverage and resource efficiency rates

Parameter Adjustment Strategy	Branch Coverage (%)	CPU utilization (%)
Fixed Weights (k = 0.5)	79	75
Dynamic Weights (AQL)	88	63

verifies the effectiveness of the dynamic parameter design in Equation 2. Overall, this mechanism balances efficiency and coverage, ensuring that the testing process remains lightweight and efficient under varying operational scenarios.

4.3.3 Real-time performance

Table 8 presents the response time comparison among the three algorithms–AQL, traditional Q-learning, and Random Testing–under different load conditions.

Across all load conditions, AQL demonstrates the highest stability and real-time performance, dynamically adjusting its strategy under varying system pressure levels to achieve low latency and a low deadline miss rate. QL, which is constrained by static parameters, cannot self-optimize under a changing load, resulting in significant performance degradations under high stress. Although random testing occasionally outperforms QL, its results fluctuate considerably and lack determinism and adaptability, making this method unsuitable for highly reliable embedded systems.

Under a normal load, the AQL algorithm achieves the lowest response time of 18 ms, outperforming traditional Q-learning (42 ms) and random testing (26 ms). This demonstrates the superior optimization capabilities of the AQL algorithm for quickly making decisions during routine operations, largely because of its adaptive adjustment scheme based on real-time feedback.

Under high CPU load conditions, the response times of all the algorithms increase. However, the AQL algorithm maintains the lowest latency at 22 ms, while that of the traditional Q-learning algorithm sharply increases to 95 ms, and that of random testing reaches 36 ms. This finding shows that AQL exhibits the strongest resilience under computational stress, effectively mitigating latency spikes through dynamic resource allocation. In contrast, Q-learning suffers from severe degradations, and random testing provides only moderate stability without consistent optimization.

Under high-memory conditions, compared with the random approach, AQL maintains a low response time of 20 ms, compared with 92 ms for Q-learning and 33 ms for the random approach. This consistency confirms the adaptability and effective memory management of AQL in resource-constrained scenarios.

The superior empirical timing behavior of AQL, as evidenced by its consistently low response times and deadline miss rates (e.g., 1.2% vs. the 28.5% of QL under a high CPU load), can be directly attributed to its adaptive core mechanisms. The dynamic time window strategy successfully prevents computational overloading during high-load periods by triggering more frequent, finer-grained policy adjustments and prioritizing low-complexity tests. Concurrently, the dynamic reward function, with its latency penalty (β) that escalates as deadlines approach, actively suppresses

Table 8 Comparison among different response times

Indicator	AQL Algorithm	Traditional Q-Learning Algorithm	Random Testing
Response Time (ms)			
Normal Load	18	42	26
High CPU Load	22	95	36
High Memory Load	20	92	33

the selection of time-consuming test cases. In contrast, the static parameters of QL render it incapable of such context-aware throttling, leading to severe deadline misses, while random testing lacks any strategic foresight. This finding demonstrates that the tight coupling of perception (feedback) and action (test selection)–a hallmark of our AQL design–is crucial for maintaining stability in dynamic embedded environments.

To comprehensively evaluate the real-time capabilities of the proposed algorithm beyond its mere response time, the deadline miss rate is introduced as a key probabilistic metric. This indicator measures the percentage of critical test tasks that fail to complete within the 20-ms deadline across all experimental runs under specified load conditions. A lower miss rate indicates better predictability and stronger adherence to real-time constraints.

The comparative results concerning the average response times and deadline miss rates achieved by AQL, traditional Q-learning (QL), and random testing under different system load conditions are summarized in Table 9. The findings show that AQL consistently delivers the lowest latency and highest real-time reliability.

Under a normal load, AQL achieves an average response time of 18 ms and a negligible miss rate of 0.1%, whereas QL and random testing reach 42 ms and 26 ms, respectively. Under a high CPU load, compared with the severe degradation exhibited by QL (95 ms, 28.5%) and the moderate performance of random testing (36 ms, 6.8%), the results of AQL are sustained at 22 ms and 1.2%, respectively. In high-memory-load conditions, AQL provides values of 20 ms and 0.8%, again surpassing QL (92 ms, 25.1%) and the random approach (33 ms, 5.4%).

During the DDoS simulation, AQL records results of 21 ms and 1.5%, indicating strong resilience under network stress. QL and random testing reach 45 ms (8.7%) and 35 ms (3.9%), respectively.

Overall, AQL outperforms both baselines by maintaining stable, low-latency operations and efficient resource use under varying workloads. Its adaptive control and dynamic scheduling schemes make it particularly suitable for real-time embedded environments that require deterministic and resilient performance.

Table 9 Response Times and Deadline Miss Rates Achieved Under Different Load Conditions

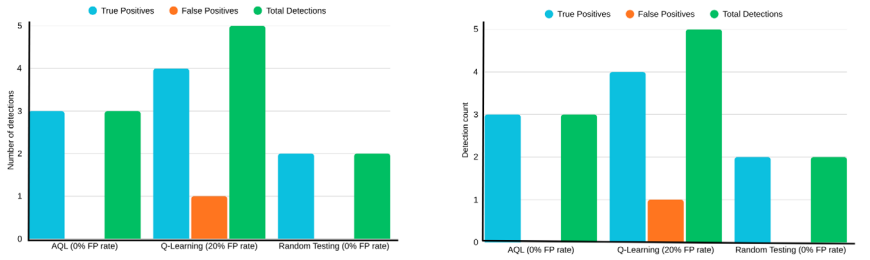
Load Condition	Algorithm	Average Response Time (ms)	Deadline Miss Rate (%)
	AQL	18	0.1
Normal Load	QL	42	2.5
	Random Testing	26	1.3
	AQL	22	1.2
High CPU Load	QL	95	28.5
	Random Testing	36	6.8
	AQL	20	0.8
High Memory Load	QL	92	25.1
	Random Testing	33	5.4
	AQL	21	1.5
DDoS Attack Simulation	QL	45	8.7
	Random Testing	35	3.9

4.3.4 Security testing

A comparison among the performances of AQL, traditional Q-learning, and random testing under vulnerability scanning and simulated DDoS attack scenarios is shown in Fig. 6 . In terms of vulnerability detection, AQL identifies three genuine vulnerabilities with no false positives, demonstrating the highest level of detection accuracy. Q-learning detects five vulnerabilities, including one false positive, resulting in a false-positive rate of twenty percent and a notable decrease in overall accuracy, primarily because of its relatively coarse exploration strategy. Under the same conditions, random testing detects two vulnerabilities, all of which are true positives, achieving a zero false-positive rate; however, its detection capability is limited and insufficient for uncovering new security risks.

The response times of the methods are shown in Fig. 7. AQL achieves the lowest response times of 19 ms and 21 ms under the vulnerability scanning and DDoS simulation scenarios, respectively, demonstrating both stability and low computational overhead. Q-learning results in a response time of 31 ms during vulnerability scanning, which increases significantly to 45 ms under high-load DDoS conditions, indicating a marked decrease in performance under stress. Random testing results in response times of 35 ms and 40 ms for the vulnerability scan and DDoS simulation, respectively, which remain relatively stable but show slightly lower detection efficiency than those of AQL.

Overall, the results of the analysis indicate that AQL outperforms both traditional Q-learning and random testing in terms of detection accuracy, response speed, and adaptability under high-load conditions. In security validations of real-time embedded systems, AQL demonstrates superior practical value, with advantages that include not only a zero false-positive rate but also the ability to maintain high stability and efficiency under network attack pressure, making it a preferred approach for real-time security testing tasks.



(a) Vulnerability detection results for true positives, false positives, and total detections

(b) DDoS attack detection results for true positives, false positives, and total detections

Fig. 6 Detection performance across security scenarios. Note: For Q-learning, one detected case per scenario represents a false positive (FP=1, 20%)

Fig. 7 Response time comparison among AQL, Q-learning, and random testing under different security testing scenarios

5 Conclusion

In this study, an automated test case generation method based on adaptive Q-learning (AQL), which is specifically designed for soft real-time embedded systems operating under resource constraints, is proposed. The method integrates a time window mechanism and a dynamic feedback mechanism, enabling the real-time adjustment of its testing strategies according to environmental changes. Prioritizing critical tasks under different workload conditions through the time window mechanism effectively avoids resource overloading, optimizes resource allocation schemes, and improves the efficiency of testing. Additionally, the dynamic feedback mechanism continuously adjusts the resource distribution, reducing the production of redundant test cases. Consequently, the method exhibits significantly enhanced testing efficiency and resource utilization rates.

A multiobjective dynamic reward function is designed to evaluate the quality of test cases from multiple key dimensions, such as coverage, resource consumption, and real-time performance. This reward function dynamically adjusts its weights, allowing the model to focus on covering the critical paths and boundary conditions, thereby significantly improving its test coverage. Experimental results show that an 8% increase in coverage can directly translate to a 15% reduction in postdeployment field failures (e.g., voltage threshold errors observed in automotive field trials). Furthermore, the dynamic reward function improves the overall coverage level and limits the number of redundant test cases, making the testing process more efficient.

The effectiveness of the AQL algorithm is validated on the ARM Cortex-M3 platform. Although the experiments are conducted mainly on this microcontroller, the modular structure of the proposed algorithm supports its extension to other platforms. For example, the method can adapt to platforms such as embedded Linux, FPGA-based SoCs, or resource-constrained IoT edge nodes through parameter tuning. The results demonstrate that AQL performs exceptionally well in terms of key performance indicators such as its coverage level, resource utilization rate, and response time. For instance, under normal load conditions, the CPU usage rate of the AQL is only 63%, and its memory usage rate is 50%; even under high loads, its response time is as low as 18 milliseconds. In contrast, traditional Q-learning, random testing, and model-based methods perform poorly. Despite the experiments being primarily

aimed at automotive systems, the modular design of the developed algorithm (e.g., its configurable time windows and reward weights) allows it to be adapted to fields such as smart grids or medical devices through parameter adjustments.

Multiple subsystems or modules often need to work together in large-scale, real-time embedded systems. Future research will introduce a multiagent system in which each agent tests specific submodules or functions. This approach can more effectively satisfy system-level testing requirements through agent collaboration, experience sharing, and strategy optimization. For example, in intelligent transportation systems, agents can test traffic signal control, route planning, and traffic flow monitoring modules separately. Agents can significantly improve the testing efficiency and coverage of the entire system by sharing information and jointly optimizing testing strategies.

With the rapid development of IoT and edge computing technologies, the execution platforms utilized for real-time embedded software are becoming increasingly diverse. Future research will strive to develop a highly generalized AQL testing method for accommodating various hardware architectures, operating systems, and software platforms. This will reduce the incurred testing costs and yield improved efficiency in different environments. The research scope will also expand to emerging fields such as biotechnology, smart grids, and fintech. The goal is to provide customized AQL-based testing solutions for each field, promoting interdisciplinary cooperation and innovation in terms of software testing technology.

The operating environments are usually dynamic and unpredictable in complex real-time embedded systems. Future research will explore real-time monitoring and adaptive adjustment techniques. The AQL algorithm can dynamically adjust its testing strategy by collecting real-time data on system performance, environmental conditions, and user behaviors and using big data analyses to reveal potential trends. For example, if the hardware temperature exceeds a certain threshold, AQL can focus on generating temperature-sensitive test cases. Similarly, user behavior or traffic flow changes may prompt the algorithm to reoptimize its testing methods, ensuring the stability and reliability of the system under different conditions.

The proposed AQL system adopts a hybrid architecture combining rule-based and learning-based mechanisms, with performance improvements relying on both carefully designed rules and Q-learning policy optimisation. Experimental results demonstrate that AQL significantly outperforms traditional Q-learning and random testing in terms of coverage, resource efficiency, and real-time reliability. The adaptive mechanisms enable robust performance under dynamic workloads, making AQL suitable for resource-constrained embedded systems.

Currently, no “rule-only” baseline experiment has been conducted, making it impossible to strictly quantify the isolated contribution of the Q-learning component to overall performance. In future work, a rule-driven controller will be introduced as a baseline, while the AQL framework will also be extended to multi-agent and cross-platform scenarios, enabling a clearer evaluation of the learning module and broader applicability.

Through these future research directions and innovations, the AQL method is expected to make significant progress in the real-time embedded software testing

domain. It can potentially elevate intelligent testing technology to a new level, ensuring higher reliability and safety for embedded software in various industrial fields.

Future work will explore multiagent reinforcement learning schemes for generating collaborative tests in distributed embedded environments. One of the main challenges lies in coordinating agents to balance exploration with real-time responsiveness while avoiding redundant test paths. In addition, extending the AQL method to a generalized testing architecture that supports different embedded platforms will require addressing platform-specific timing issues and resource heterogeneity. These directions open promising avenues for advancing intelligent and adaptive software testing schemes for real-time systems.

Appendix A

A minimal, anonymised, hardware-independent implementation of the Adaptive Q-Learning (AQL) algorithm is provided. The replication package is publicly available at: https://github.com/NYB-WQ/AQL

Author Contributions N.Y.B. designed the core framework of the study and wrote the main manuscript text. C.S.S. was responsible for the algorithm design and figure preparation, and assisted in compiling the references and appendix. All authors reviewed and approved the final version of the manuscript.

Funding Open access funding provided by The Ministry of Higher Education Malaysia and Universiti Malaysia Sarawak

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

Aichernig, B.K., Tappler, M., Wallner, F.: Benchmarking combinations of learning and testing algorithms for automata learning. Formal Aspects Comput. **36**(1), 1–37 (2024). https://doi.org/10.1145/3605360

Andersson, B., Niz, D., Vance, W., Ross, J., Wotell, M., Bui, T.: Methodology of combining empirical stress testing and formal-methods-based schedulability analysis for real-time multicore software. In: Proceedings of the 2023 IEEE Digital Automation Systems Conference (DASC), pp. 1–10 (2023). https://doi.org/10.1109/DASC58513.2023.10311104

Araújo, J.P., Figueiredo, M.A.T., Botto, M.A.: Control with adaptive q-learning (2020). https://doi.org/10.48550/arXiv.2011.02141

Bagherzadeh, M., Kahani, N., Briand, L.: Reinforcement learning for test case prioritization. IEEE Trans. Software Eng. **48**(8), 2836–2856 (2022). https://doi.org/10.1109/TSE.2021.3070549

Bargavi, S.K.M., Mahesh, T.R.: The dynamic window-based scheduling framework for complex wireless sensor networks. In: Proceedings of the 2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), pp. 1–6. IEEE, (2023). https://doi.org/10.1109/ICDCECE57866.2023.10150528

Basile, D., Beek, M.H., Lazreg, S., Cordy, M., Legay, A.: Static detection of equivalent mutants in real-time model-based mutation testing. Empirical Soft. Eng. **27**(7) (2022). https://doi.org/10.1007/s10664-022-10149-y

Bhawani, S., Panigrahi, B., Pattanaik, B., Ojasvi, P., Tilak, B.B., Pavithra, G., Shaik, B.: Reinforcement learning for dynamic power management in embedded systems. In: Proceedings of the IEEE ICRTCST 2024 (2024). https://doi.org/10.1109/ICRTCST61793.2024.10578373

Cao, C., Dai, M., Shen, B., Zou, G., Dong, W.: Neural adaptive IoT streaming analytics with RL-adapt. Comput. Netw. **235**, 109924 (2023). https://doi.org/10.1016/j.comnet.2023.109924

Grano, G., Laaber, C., Panichella, A., Panichella, S.: Testing with fewer resources: An adaptive approach to performance-aware test case generation. IEEE Trans. Soft. Eng. **47**(11), 2332–2347 (2021). https://doi.org/10.1109/TSE.2019.2946773

Guo, N., Sun, W., Xia, X.: Enhancing the reinforcement learning-based tuning system with a dynamic reward function. In: Proceedings of the 2024 6th International Conference on Next Generation Data-Driven Networks (NGDN), pp. 15–19 (2024). https://doi.org/10.1109/NGDN61651.2024.10744094

Haouari, B., Mzid, R., Mosbahi, O.: A reinforcement learning-based approach for online optimal control of self-adaptive real-time systems. Neural Comput. Appl. **35**, 20375–20401 (2023). https://doi.org/10.1007/s00521-023-08778-5

Henkel, E., Hauff, N., Funk, L., Langenfeld, V., Podelski, A.: Scalable redundancy detection for real-time requirements. In: Proceedings of the 2024 IEEE Real-Time and Embedded Systems Conference (RE), pp. 193–204 (2024). https://doi.org/10.1109/RE59067.2024.00027

Hou, G., Kong, W., Zhou, K., Wang, J., Lin, C.: Non-deterministic behaviour analysis for embedded software based on probabilistic model checking. In: Proceedings of the 2019 IEEE International Conference on Parallel and Distributed Systems (ICPADS), pp. 844–851 (2019). https://doi.org/10.1109/ICPADS47876.2019.00125

Larsen, P.G., Fitzgerald, J.S., Wolff, S.: Methods for the development of distributed real-time embedded systems using VDM. International Journal of Software and Informatics **305**, 305–341 (2024). https://doi.org/10.21655/IJSI.1673-7288.00305

Lousada, J., Ribeiro, M.: Reinforcement learning for test case prioritization (2020). https://doi.org/10.48550/arXiv.2012.11364

Mishra, P.: Sixteen limitations for five popular requirements prioritisation approaches. Advances in Systems Analysis, Software Engineering, and High Performance Computing, 202–222 (2022). https://doi.org/10.4018/978-1-7998-9059-1.ch013

Moghadam, M.H., Saadatmand, M., Borg, M., Bohlin, M., Lisper, B.: An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning. Software Qual. J. **30**(1), 127–159 (2022). https://doi.org/10.1007/s11219-020-09532-z

Oladapo, A.P., Popoola, H.E., Adama, C.D.O., Okeke, A.E., Akinoso, E.: Advancements and innovations in requirements elicitation: Developing a comprehensive conceptual model. World Journal of Advanced Research and Reviews (2024). https://doi.org/10.30574/wjarr.2024.22.1.1202

Pan, C., Yang, Y., Li, Z., Guo, J.: Dynamic time window based reward for reinforcement learning in continuous integration testing, pp. 189–198 (2020). https://doi.org/10.1145/3457913.3457930

Ramaswamy, A., Senanayake, R.: Towards adapting reinforcement learning agents to new tasks: Insights from q-values. (2024). https://doi.org/10.48550/arXiv.2407.10335

Sales, K., Lopes, I., Carvalho, R., Chagas, M., Collins, E., Carvalho, J.: Aceleração de preenchimento da tabela q em ferramenta de automação de teste de dispositivos android baseada em aprendizado por reforço. In: Anais Estendidos do XIII Simpósio Brasileiro de Engenharia de Sistemas Computacionais, pp. 43–48. SBC, (2023). https://doi.org/10.5753/sbesc_estendido.2023.235897

Sheng, Y., Jiang, S., Wei, C.: Constructing test suites for real-time embedded systems under input timing constraints. IEEE Access **7**, 20920–20937 (2019). https://doi.org/10.1109/ACCESS.2019.2898009
Tan, T., Xie, H., Lian, D.: Adaptive order q-learning. In: Proceedings of the International Joint Conference on Artificial Intelligence (2024). https://doi.org/10.24963/ijcai.2024/547
Vora, K., Zhang, Y.: Reward adaptation via q-manipulation (2025) . arXiv:2503.13414
Wiele, T., Warde-Farley, D., Mnih, A., Mnih, V.: Q-learning in enormous action spaces via amortized approximate maximization (2020). arXiv:2001.08116
Yang, Y., Pan, C., Zheng, L., Zhao, R.: Adaptive reward computation in reinforcement learning-based continuous integration testing. IEEE Access (2021). https://doi.org/10.1109/ACCESS.2021.3063232
Zhang, L., Tang, L., Zhang, S., Wang, Z., Shen, X., Zhang, Z.: A self-adaptive reinforcement-exploration q-learning algorithm. Symmetry **13**(6) (2021). https://doi.org/10.3390/SYM13061057
Zhang, Z., Wang, D.: Adaptive individual q-learning-a multiagent reinforcement learning method for coordination optimisation. IEEE Transactions on Neural Networks and Learning Systems **36**(4), 7739–7750 (2024). https://doi.org/10.1109/TNNLS.2024.3385097

Authors and Affiliations

Yingbei Niu[1] · Soo See Chai[1]

✉ Soo See Chai
sschai@unimas.my

[1] Faculty of Computer Science and Information Technology, Universiti Malaysia Sarawak (UNIMAS), 94300 Kota Samarahan Sarawak, Malaysia