Optimized Test Scenario Identification Method based on Refined Warshall's Dynamic Programming for Validating Reinforcement Learning Model

Janghwan Kim[1], and R. Young Chul Kim[1*]
[1] Department of Software and Communications Engineering, Hongik University, Sejong, 30016, South Korea
[e-mail: lentoconstante@hongik.ac.kr, bob@hongik.ac.kr*]
*Corresponding author: R. Young Chul Kim

Received November 11, 2025; revised February 22, 2026; accepted March 24, 2026; published April 30, 2026

Abstract

In recent decades, Reinforcement learning (RL) has achieved remarkable performance in sequential decision-making tasks; however, validating RL-based software remains a significant challenge due to the exponential growth in the number of state–action combinations. We propose an optimized test-scenario identification method for a formal validation mechanism that integrates a refined Warshall's dynamic programming algorithm and path optimization to ensure efficient test coverage for RL systems. The method constructs a directed abstract graph from the RL model, applies transitive closure analysis to check reachability between states, and identifies missing states/transitions before testing. Our goal is to use test-scenario optimization to generate a minimal yet sufficient set of test cases, thereby achieving maximum coverage with minimal redundancy. This approach reduces verification complexity while maintaining mathematical rigor, making it well-suited for safety-critical applications such as autonomous driving. The proposed mechanism provides a scalable, interpretable validation process, offering a foundation for the reliable deployment of RL–based software in real-world systems.

Keywords: Reinforcement Learning, Test Scenario Optimization, AI Software Validation

A preliminary version of this paper was presented at APIC-IST 2025 and was selected as an outstanding paper.

http://doi.org/10.3837/tiis.2026.04.030 **ISSN : 1976-7277**

1. Introduction

As artificial intelligence (AI) technologies expand into safety-critical domains such as autonomous driving, robotics, and medical systems, the importance of software verification techniques that ensure reliability and safety is increasingly emphasized. RL, in particular, demonstrates exceptional performance in addressing complex sequential decision-making problems and has garnered significant attention. However, its internal structure remains a "black box," making it difficult to explain the logical basis of its decisions clearly. This opacity poses a substantial barrier to establishing the trustworthiness of RL-based software in real-world applications that require guaranteed safety, stability, and robustness [1-2].

Recent studies have proposed various approaches to address verification challenges in AI systems. A comprehensive analysis of key issues in AI software quality has been conducted, while the necessity of testing for AI-based software has been highlighted [3-4]. A verification framework for RL policies in autonomous driving environments has been introduced, and probabilistic safety verification techniques have been employed to evaluate policy stability [5-6]. Additionally, the concept of operational profile-based validation for determining test priorities has been presented, and scenario-based testing has been utilized to enhance the structural efficiency of model verification [7-8]. Despite these efforts, most studies focus on supervised or unsupervised learning models, and a formal verification framework that adequately accounts for the dynamic state–action transition structure of RL remains insufficiently established.

The fundamental difficulty in RL verification stems from the combinatorial explosion of the state–action space. For instance, in a tree structure of depth k where each state connects to n actions and each action leads to n new states, the number of possible state transitions grows to n^k. As the number of states increases, the required test cases escalate exponentially, rendering exhaustive verification practically impossible. Consequently, RL system safety validation faces a time–cost trade-off, necessitating an efficient, mathematically sound, abstracted verification procedure [9-10].

Existing studies primarily focus on empirical, scenario-based testing or probabilistic exploration-based validation, with relatively limited attention to graph-theory-based formal verification. In particular, few studies address both transition reachability and test scenario optimization simultaneously. Moreover, conventional test optimization techniques often rely on heuristics or empirical rules, failing to guarantee the logical completeness of the verification target [11].

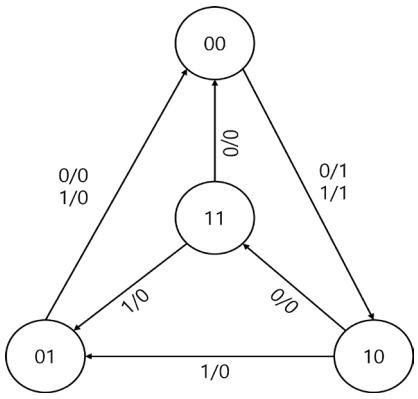
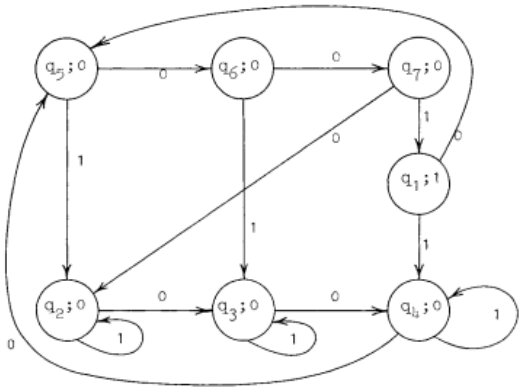
To overcome these limitations, we propose a validation mechanism for RL software based on Warshall's dynamic programming algorithm. The proposed method transforms the complex state–action graph of an RL model into a finite, directed, abstract graph, formally verifies reachability via transitive closure, and computes optimal test scenario paths with minimal test cases using Dijkstra's algorithm. By achieving maximum coverage without exhaustive testing, this approach aims to establish a formal and efficient validation framework for RL software.

2. Related Works

Moore and Mealy's Finite State Machine: Finite State Machines (FSMs), which form the foundation of digital systems and computation theory, are abstract models with a limited number of states, transitions between those states based on input, and an output. The two most representative models of FSMs are the Mealy Machine, proposed by George H. Mealy in 1955,

and the Moore Machine, proposed by Edward F. Moore in 1956. The two models differ fundamentally in how they determine their outputs, which directly affects the characteristics and performance of a digital circuit design.

Table 1. Mealy and Moore's Finite Machine

Mealy Machine	Moore Machine
	
A Mealy machine is a 6-tuple (S, s_0, $\sum$, Λ, T, G) where each component is defined as follows: *(1) S: a finite set of states* *(2) s_0: a start state (or initial state), where $s_0 \in S$* *(3) $\sum$: a finite set called the input alphabet* *(4) Λ: a finite set called the output alphabet* *(5) T The transition function, which maps a pair of (current state, input symbol) to the next state, T: $S \times \sum \rightarrow S$* *(6) G The output function, which maps a pair of (current state, input symbol) to an output symbol, G: $S \times \sum \rightarrow \Lambda$*	A Moore machine is a 6-tuple (S, s_0, $\sum$, *O* , δ, G) where each component is defined as follows: *(1) S: a finite set of states* *(2) s_0: a start state (or initial state), where $s_0 \in S$* *(3) $\sum$: a finite set called the input alphabet* *(4) O: a finite set called the output alphabet* *(5) δ: The transition function (delta), which maps a pair of (current state, input symbol) to the next state, δ: $S \times \sum \rightarrow S$* *(6) G The output function, which maps a pair of (current state, input symbol) to an output symbol, G: $S \rightarrow O$*

The Mealy Machine was first introduced in the 1955 paper, "A Method for Synthesizing Sequential Circuits."[10] Its most significant characteristic is that the current state and the current input determine the output. Because of this, the output reacts immediately to changes in the input, giving it a very fast response time. Furthermore, when implementing the same function, a Mealy Machine can generally be designed with fewer states than a Moore Machine, which can be advantageous for circuit optimization. However, because the output depends directly on the input, it also has a disadvantage: temporary noise in the input signal can directly affect the output, potentially compromising the system's stability.

On the other hand, the Moore Machine, introduced in the 1956 paper "Gedanken Experiments on Sequential Machines," determines its output based only on the current state [12]. The input's role is only to determine the next state; it has no direct influence on the output. Thanks to this characteristic, the output is synchronized with the clock signal, giving it the significant advantage of being free from input signal noise and highly stable. This also makes system design and debugging easier. However, because the output is generated after the state has changed, its response to an input change is delayed by at least one clock cycle compared to a Mealy Machine. Additionally, implementing the same function may sometimes require more states. **Table 1** shows a comparison between Mealy and Moore's machines.

Mealy Machines and Moore Machines exhibit clear differences in their outputs, timing, and stability. The Mealy Machine is suitable for systems where fast response time is critical, and state minimization is required. The Moore Machine is better suited to systems where output stability and design simplicity are prioritized.

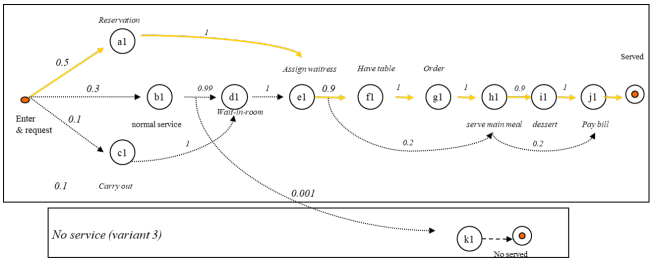
This approach is conceptually derived by adapting the formal state-transition and output-generation paradigms of the classic Mealy and Moore models to define and test the complex behaviors of RL agents in structured scenarios.

Scenario-based Integration Testing [8]: In object-oriented software development, the adaptive use case methodology improves the efficiency of scenario-based integration testing by integrating the software design, development, and testing processes via a series of algorithmic transformations that generate test plans at the design stage [8].

Index	Action Unit Name
A1	Action 1
B1	Action 2
C1	Action 3
D1	Action 4
⋮	⋮

	Use Case Scenarios \ Action Units	A1	B1	C1	D1	...
A Scenario	Scenario 1	1	3	4	5	...
	Scenario 2	1		2	3	...
	Scenario 3		1		2	...

Fig. 1. Use case Action Matrix



Weighted value: 1 1 1 1 1 1 1 1 1 1 1 = 11 $\prod \frac{Si*Pi}{P}$

	a1	b1	c1	d1	e1	f1	g1	h1	i4	j1	k1	Weighted value amount	Total amount of probability of occurrence
Main path (Reserv.)	1				2	3	4	5	6	7		**7**	*0.5*1 * 0.9*1 * 0.9*1=**0.405***
Variant 1 (normal)		1		2	3	4	5	6	7	8		**8**	*0.3*1*0.9*1 * 0.9* 1 =**0.243***
Variant 2 (Carryout)			1	2	3			4		5		**5**	*0.1*1 * 0.1*1 * 0.1*1=**0.001***
Variant 3 (no service)		1									2	2	*0.3*1 * 0.001*1 =**0.0003***

Fig. 2. Action Matrix for Test Plan

This clarifies the execution path by numerically indicating specific operational steps within a scenario. **Fig. 1** is an example of a Use Case Action Matrix. Based on the list extracted from **Fig. 1**, an Action Matrix is created, and the Graph is verified and improved as we show in **Fig. 2**. Additionally, applying Musa's operational profile concept to optimize the scenario execution order enhances test productivity by utilizing software test metrics and effectively provides test prioritization at the design stage [7].

The Original Markovian Decision Process [13]: Bellman demonstrated that decision-making problems could be expressed as a form of recursive optimization using a recurrence relation [13]. Bellman proved that this recurrence relation achieves asymptotic linear convergence under the condition that the matrix is positive, thereby providing a mathematical proof of the principle of optimality.

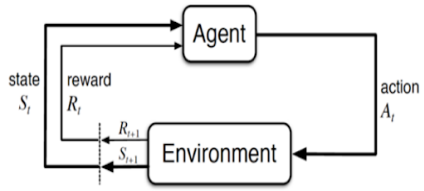


Fig. 3. Markov Decision Process

The Markov Decision Process (MDP) is a framework for modeling sequential decision-making problems and serves as the theoretical foundation for RL. An MDP consists of five core elements: a set of states (S), a set of actions (A), transition probabilities (P), rewards (R), and a discount factor (γ). **Fig. 3** illustrates the Markov Decision Process [9].

(1) $MDP = (S, A, P, R, \gamma),\ s \in S,\ a \in A,\ (0 \leq \gamma \leq 1),$

(2) $P(s'|s,a)=P[S_{t+1}=s'|S_t=s,\ A_t=a]$

(3) $R(s,a)=E[R_{t+1}|S_t=s,\ A_t=a]$

When an agent takes an action (A_t) in a specific state (S_t), the environment transitions to a new state (S_{t+1}) and provides a corresponding reward (R_{t+1}) to the agent. At this time, the next state (S_{t+1}) depends only on the current state (S_t) and the action (A_t), following the Markov Property, which states it is independent of past states. The discount factor (γ) represents the present value of future rewards and has a value between 0 and 1. This value determines whether the agent places a greater emphasis on immediate rewards or future rewards. As γ approaches 0, the agent focuses more on immediate rewards, while as it approaches 1, it considers future rewards from a long-term perspective.

Recent Trends in RL Validation and Limitations:

Table 2. Comparison of Diverse Software (SW) Validation

Features	Traditional SW Validation	Recent RL Validation Researches	Our Proposed Method
Focus	Random / Simulation Testing	Adversarial / Fuzzing Testing	Optimized Scenario Identification
Approach	Stochastic Sampling (Random inputs)	Optimization-based Search (gradient/Heuristic)	Graph-based Reachability (Refined Warshall)
Goal	General Performance Evaluation	Finding specific counter-examples (Bugs)	Systematic Identification of Test Scenarios
Completeness	Low (Misses Rare events)	Probabilistic (Cannot guarantee coverage)	Logically Complete (Guaranteed Reachability)

Efficiency	Very Low (Requires Massive iteration)	Medium (Iterative search Process)	High (Minimal Test Set via Optimization)
Interpretability	Black box	Black box (Focus only on failure cases)	Gray box (Explicit State-Action Transitions)
SOTIF supported	Limited	Focuses on 'Unsafe' regions only	Covers both 'Safe' & 'Unsafe' Transition logic

While foundational theories like FSM and MDP provide the structural basis for modeling, recent advancements in RL validation have largely focused on Adversarial Testing or Fuzzing techniques [14]. These methods aim to identify failure cases by adding perturbations to the input space or heuristically searching for corner cases.
However, as summarized in Table 2, these approaches are inherently probabilistic and often suffer from redundant testing without guaranteeing logical coverage of the scenario space. They excel at finding specific bugs but fail to provide a systematic assurance of safety required for Safety of the Intended Functionality (SOTIF).
In contrast, our proposed method focuses on 'Test Scenario Identification'. By applying the Refined Warshall Algorithm to an abstracted state model, we ensure logical completeness of the validation scenarios before execution.

3. Research Background

Deep Reinforcement Learning (DRL) can achieve impressive results, often matching or surpassing human performance in complex decision-making tasks [15]. However, the Deep Neural Network (DNN) in DRL acts like a "black box," making it hard to understand its decisions. This lack of clarity is a major issue for safety-critical systems, such as autonomous vehicles or medical devices, where safety, stability, and reliability are essential.
One key challenge in testing DRL systems is scalability. DRL agents explore numerous states and actions, leading to an enormous number of possible combinations. For instance, **Fig. 4** shows the number of test scenarios of the DRL system. If each state connects to $n$actions, and each action leads to n new states, the total number of paths in a sequence of k steps becomes n^k. As n and k grow, the number of test cases explodes, making it impossible to test every scenario. This creates a safety validation gap, making it tough to confirm whether a DRL policy works correctly in normal conditions or handles abnormal situations safely. To address this, we need a simplified way to represent the DRL model by grouping similar states and streamlining transitions.

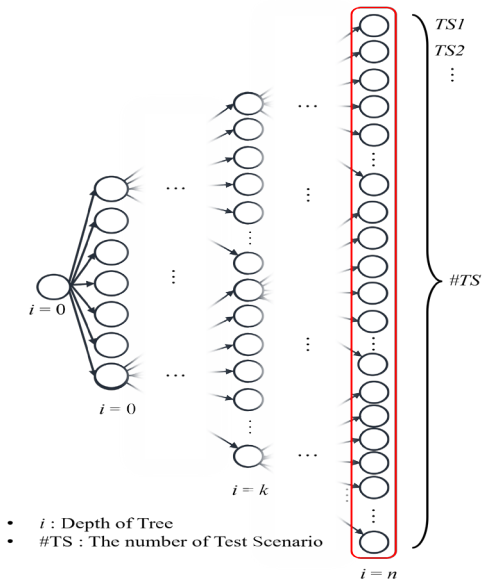


Fig. 4. State Space Explosion of RL Software

4. RL Model Validation Mechanism

The previous Model-Based Testing (MBT) is a mature software engineering methodology for testing complex systems [16]. MBT operates by generating test cases from an abstract model of system requirements rather than the code itself. We apply to the challenge of RL validation with the core benefits of MBT—early defect detection, automated test case generation, improved test coverage, and reduced maintenance costs. Our proposed method extends the new RL context with a well-established testing paradigm. We describe a four-step scenario-based testing methodology to validate a software system embedded with an RL model.

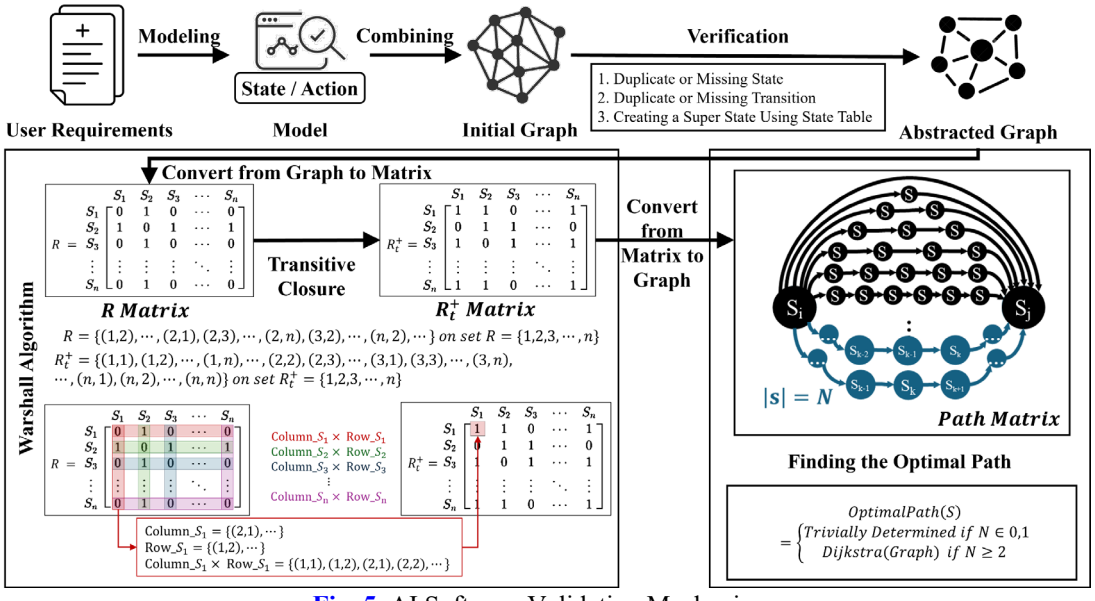


Fig. 5. AI Software Validation Mechanism

Fig. 5 shows the four-step methodology proposed in this paper, which integrates several concepts. It consists of a consistent pipeline that transforms high-level system requirements into a single, optimized, and formally verifiable test case. This approach is proposed based on the following factors:

Abstraction: Complexity is managed through abstraction techniques borrowed from Model-Based Testing (MBT) and Hierarchical Reinforcement Learning (HRL). MBT abstracts the system's extensive internal code and variables into a concise model of the core functional flow based on requirements. To achieve this, the system is structured into states, transitions, and events, and state-space reduction is applied [17]. Millions of lines of code and infinite variable combinations are eliminated, and the model is compressed to only the core states meaningful to the user (e.g., "before login," "login success," "shopping cart," "payment completion"). By automatically generating test cases that cover all paths based on the state transition diagram constructed this way, efficiency is increased over manual enumeration, and the risk of missing scenarios is eliminated.

Formal Verification: Ensures testability through graph reachability analysis. Before generating optimal test paths, this step mathematically proves the logical existence of a path from the starting point to the goal. This process preemptively screens out infeasible test scenarios. It formally verifies the absence of design flaws or errors in the abstract model itself, which serves as the basis for the tests. This ensures the reliability and validity of the subsequent test generation process.

Optimization: Ensures test efficiency using a shortest-path algorithm. This involves automatically finding the single, most efficient (lowest cost) optimal test case among numerous possible test scenarios that reach the system's starting state and the target state, based on defined criteria such as time, cost, and steps. This optimization process enables verification of the system's core functionality with minimal effort, much like a navigation system finding the fastest route. **Table 3** shows descriptions that are used in the following equations.

Table 3. Symbol Description of Equations

Symbol	Definition	Description
$\boldsymbol{S = \{S_1, S_2 \cdots, S_n\}}$	Set of States	A finite set representing all possible abstract states of the system. Each $\boldsymbol{S_i}$ denotes a specific situation (e.g., lane keeping, approaching an intersection).
$\boldsymbol{A = \{A_1, A_2 \cdots, A_n\}}$	Set of Actions	A finite set of all possible actions executable in each state
$\boldsymbol{\delta: S \times A \rightarrow S'}$	Transition Function	A deterministic function that returns the next state for a given state–action pair. $\boldsymbol{S_k = \delta(S_i, a_j)}$ denotes a state transition.
$\boldsymbol{S_i}$	Current State	The abstract representation of the system's current situation.
$\boldsymbol{a_j}$	Action Performed	The semantic decision or control action taken in the current state.
$\boldsymbol{S_k}$	Resulting State	The next state was reached after performing the action.
$\boldsymbol{G(a_j)}$	Action-based Transition Set	The set of all transitions triggered by the same action $\boldsymbol{a_j}$. $G(a_j) = \{(S_i, S_k) \mid \delta(S_i, a_j) = S_k\}$.
$\boldsymbol{S^*}$	Set of Super States	A higher-level set of abstracted states obtained by merging repetitive transition patterns.
$\boldsymbol{S^*_{new}}$	Super State for new	The union of all states represents controlled deceleration and safe stopping transitions.
⋮	⋮	⋮

Our approach provides a step-by-step process for generating reliable, efficient test cases from software requirements. First, requirement-based abstract modeling is performed to extract the essential state and action information needed for test case generation. Next, an initial graph is constructed and verified by experts to identify any duplicate or missing states and transitions (actions). Based on this verification, related states are combined to define "super states," and a refined abstract graph is produced. Then, Warshall's dynamic programming technique is applied to verify the transition closure, a prerequisite for test feasibility. Finally, Dijkstra's algorithm is used to find optimal paths across the test cases, enabling the creation of efficient, minimal test scenarios.

4.1 Requirement-Based Abstract Modeling

The first step of the proposed method is to create an initial graph. In this step, the large, continuous state space of the RL agent is refined into a limited set of key states (nodes) and actions (edges) based on domain expert knowledge. The domain expert extracts these states and actions from the System Requirements Specification (SRS). Since the RL agent's decision space is almost infinite, it is abstracted into a meaningful finite unit to achieve the goal of "maximum coverage with minimum test cases." Through this process, the SRS is transformed into a testable graph model. This process can be divided into the following steps.

Key Requirement Identification: The domain expert first identifies the main functional and non-functional requirements from the SRS, especially those related to safety-critical scenarios. For example, a clear, testable scenario might be: "The autonomous car must stop at a red light."

State Extraction through Abstraction: The expert does not handle raw state data directly (for example, sensor data, GPS coordinates (x, y, z), or vehicle speed). Instead, the expert defines abstract states based on conditions or predicates related to the requirements. This process divides the infinite continuous state space into a finite number of logical regions. For instance, the state of an autonomous vehicle can be abstracted as "lane keeping," "approaching intersection," "stopped," or "collision risk." **Table 4** shows an example of abstract states derived from requirements.

Table 4. Example of Abstract States Derived from Predicates (Autonomous Vehicle Case)

States	Node Name	Predicate Condition	Description
S_1	Lane Keeping Clear	(is_lane_keeping=True) ∧ (is_approaching_intersection=False) ∧ (is_collision_risk=False)	Normal lane-keeping while no intersection or collision risk exists.
S_2	Approaching Intersection	(is_lane_keeping=True) ∧ (is_approaching_intersection=True) ∧ (is_stopped=False)	The vehicle is approaching an intersection while staying in its lane.
S_3	Stopped At Intersection	(is_approaching_intersection=True) ∧ (is_stopped=True)	The vehicle stopped at the intersection.
S_4	Collision Risk Active	(is_collision_risk=True)	The vehicle detects a collision risk with the front vehicle.

S_5	Lane Departure	(is_lane_keeping=False)	The vehicle has deviated from the lane.
⋮	⋮	⋮	⋮

Action Extraction through Abstraction: After defining the abstract states, the next step is to identify the causes of transitions between these states—called Abstract Actions. These actions are not low-level control outputs of the RL agent (e.g., steering by 3.5°, or applying 20% brake pressure), but rather semantic decisions that lead to changes in abstract states. **Table 5** summarizes the abstract actions.

Table 5. Abstract Actions for the Lane-Keeping and Intersection Scenario

Actions	Action Name	Description
A_1	Follow Lane	The vehicle maintains its current lane while monitoring lane boundaries and traffic signs.
A_2	Apply Brake	The vehicle slows or stops when approaching an intersection or an obstacle.
A_3	Detect Obstacle	The system identifies an obstacle or a slow-moving front vehicle within its detection range.
A_4	Emergency Brake	Immediate braking action triggered to prevent a collision in a high-risk situation.
A_5	Fail to Steer	Represents a failure to maintain steering control, resulting in lane departure.
⋮	⋮	⋮

Initial Abstraction Graph Generation: This step creates an initial directed graph of $G = (S, A)$. Here $S = \{S_1, S_2, \dots, S_n\}$ is the set of abstract states, and $A = \{A_1, A_2, \dots, A_m\}$ is the set of abstract actions. Each directed edge $e = (S_i, A_k, S_j) \in E$ shows a transition from state S_i to state S_j via action A_k. The graph clearly represents the system's requirements in a simple, finite way for the RL agent's environment. It includes normal paths that meet requirements and abnormal paths showing failures or violations. This procedure takes the initial graph $G = (S, A, \delta)$ as input and, based on the requirement specification, integrates repetitive state sequences and supplements missing transitions to generate a refined abstract graph $G' = (S', A', \delta')$.

First, using the set of repetitive transitions, it identifies consecutive transitions with the same purpose and merges them into a single superstate. During merging, all edges "entering or exiting" the sequence are reconfigured based on the super state, and internal edges within the sequence are removed. The refined abstract graph is then reconstructed, incorporating the merged superstates and existing states. This initial graph G serves as the foundation for further validation and test-case generation. **Table 6** shows an example of Abstract Actions and State Transition for the Lane-Keeping and Intersection scenario.

Table 6. Abstract Actions and State Transitions for the Lane-Keeping and Intersection Scenario

Current States	Action	Next States	Description
S_1	A_1	S_1	Normal lane-keeping while no intersection or collision risk exists.
S_1	A_1	S_2	The vehicle is approaching an intersection while staying in its lane.
S_2	A_2	S_3	The vehicle is stopped at the intersection.

S_1	A_3	S_4	The vehicle detects a collision risk with the front vehicle.
S_4	A_4	S_3	The vehicle has deviated from the lane.
S_1	A_5	S_5	Lane departure occurs (violation of the requirement).
⋮	⋮	⋮	⋮

Refinement of the Initial Graph: In this step, the initial graph from the previous step is verified and refined. This step is not just for reducing the number of states and edges. The main goal is to check the logical consistency and requirement alignment of the graph, while grouping repeated state–action patterns into higher-level units called Super States [18]. In **Table 5**, some abstract actions (such as A1(follow lane), A2(apply brake), A3(detect obstacle), A4(emergency brake), A5(fail to steer)) appear many times across different driving scenarios.

$$S_{\text{Approaching Intersection}} \xrightarrow{a_{\text{apply\_brake}}} S_{Stopped\ at\ Intersection}$$

For example, a state-action-state combination is shown in **Fig. 5**. This pattern, meaning "approach and stop at an intersection," often appears in other contexts, such as traffic-light stops or pedestrian detection. Therefore, it can be grouped into one such as, S_{new} named "Safe Stop" as a superstate.

$$S_{\text{Lane Keeping Clear}} \xrightarrow{a_{\text{detect obstacle}}} S_{\text{Collision Risk Active}} \xrightarrow{a_{\text{emergency\_brake}}} S^*_{\text{Emergency Stop}}$$

To ensure system safety in abnormal control situations, this study defines an emergency stop scenario as a high-level abstract state called Super State Emergency Stop. Unlike regular deceleration or a safe stop, this state covers situations requiring immediate braking due to unpredictable internal errors or external collision risks.
Formally, a subset of the system's state set S, representing abnormal conditions, is defined as follows.

$$S_{Abnormal} = \{S_{Collision\ Risk\ Active}, S_{Lane\ Departure}\} \subseteq S$$

Emergency braking of actions commonly performed in abnormal conditions with $a_{Emergency\ Break}$, the transition function is defined as follows:

$$\delta(S_i, a_{Emergency\ Brake}) = S_{Stopped\ At\ Intersection}, \forall S_i \in S_{Abnormal},$$

The set of transitions induced by the same behavior is as follows:

$$G(a_{Emergency\ Brake}) = \{ (S_i, a_{Emergency\ Brake}) \mid S_i \in S_{Abnormal} \},$$

Accordingly, new Super State which is $S^*_{Emergency\ Stop}$ is defined as an abstraction of the following action-outcome unit:

$$S^*_{Emergency\ Stop} = \cup\{S_i\} = \{S_{Collision\ Risk\ Active}, S_{Lane\ Departure}\}$$

Thus, the new Super State $S^*_{Emergency\ Stop}$ is a higher-level grouping of repeated transitions from abnormal states ($S_{Abnormal}$) to a stopped state $S_{Stopped\ At\ Intersection}$ via an emergency brake action $a_{Emergency\ Brake}$. This abstraction reduces the number of graph nodes and edges ($|S^*| \ll |S|, |\delta^*| \ll |\delta|$) and significantly decreases the number of test scenarios ($T^* \propto |S^*| \times |A| \ll T$).
After deriving the super states Safe Stop and Emergency Stop, the model replaces their state and transition subsets in the initial graph to create a refined abstract graph. The goal is to

eliminate redundancy and incomplete state-action transitions in the initial graph, reorganizing the system's behavior into a more consistent structure. Specifically, the initial graph may have redundant states with the same function or missing action transitions for certain requirements, which could disrupt consistency [19]. These issues are resolved and unified by replacing the superstates. The refined graph is much simpler and more logically cohesive than the initial graph. Each superstate directly represents the system's semantic behavior at a higher level, without relying on detailed sub-transitions. As a result, the graph eliminates redundancy, restores missing transitions, and evolves into a state that improves both requirement conformance and test efficiency.

4.2 Refined Warshall's Dynamic Programming

In this step, the goal is to ensure that the test scenarios are logically reachable before performing any optimization or path search. Since our system includes an RL model defined by an MDP, the number of possible state–action combinations can theoretically approach infinity. However, testing every possible case is computationally impossible. Therefore, a formal process is required to confirm that at least one valid path exists from the initial state S_0 to the target state S_{goal} before executing any tests.

To achieve this, the Warshall algorithm is applied to compute the transitive closure of the abstract graph created in earlier stages [20]. This calculation determines whether there exists a path between any two states in the graph. If S_{goal} is unreachable from S_0, it indicates that either the system design or the abstract model contains a logical gap. Thus, the transitive closure serves as a formal pre-validation step, ensuring the model's completeness and consistency. This analysis also provides a practical advantage for testing. By identifying reachable and unreachable state pairs, we can selectively prioritize meaningful test cases while eliminating redundant or impossible ones. For example, if a transition between two states does not exist, testing that scenario is unnecessary. Conversely, if several valid paths exist, we can focus on the most critical or high-risk ones. This selective approach improves testing efficiency by reducing test cases while maintaining high coverage across significant behavioral patterns.

Algorithm 1. Optimized Test Scenario Identification

Input: System Requirement Specification
$G_{raw}(S, A)$: Initial State-Action Graph
$W = \{w_1, w_2, w_3\}$: eights for Hybrid Cost Function
Output: P_{opt}: Set of Optimized Test Scenarios
Initialize:

Initialize $G_{abstract} \leftarrow G_{raw}$
/ Step 1: Abstraction & Refinement /
Identify repetitive sequences in S based on SRS
For each sequence $seq \in S$:
Merge seq into Super State S^*
Update $G_{abstract}$ with S^*
/ Step 2: Reachability Analysis /
Compute Adjacency Matrix M from $G_{abstract}$
Calculate Transitive Closure $R = Warshall(M)$
If $R[Start, Goal] \neq 1$ **Then**
    Return "Error: Goal Unreachable (Logical Defect)"

12 / Step 3: Path Optimization /
13 **For each** edge $e \in G_{abstract}$:
14 Calculate Cost $C(e) = w_1 \cdot C_{novel} + w_2 \cdot C_{risk} + w_3 \cdot C_{req}$
15 $P_{opt} \leftarrow Greedy\_Dijkstra(G_{abstract}, Start, Goal, C)$
16 **Return** P_{opt}

The overall procedure of the proposed method is formally described in Algorithm 1. The process takes the System Requirement Specification (SRS) and the initial state-action graph (G_{raw}) as inputs and outputs a set of optimized test scenarios (P_{opt}). The algorithm consists of three main phases: First, in the Refinement Phase (Lines 2-6), the algorithm reduces the complexity of the state space. It identifies repetitive action sequences (e.g., continuous braking) based on the SRS and merges them into abstracted 'Super States' (S^*). This step is crucial for preventing the state explosion problem inherent in RL models. Second, in the Reachability Verification Phase (Lines 7-11), the algorithm validates the logical completeness of the refined graph. By computing the transitive closure (R) using Warshall's algorithm, it mathematically verifies whether a valid path exists from the initial state to the target state. If the target is unreachable ($R[Start, Goal] \neq 1$), the process terminates and reports a logical defect in the model design. Finally, in the Path Optimization Phase (Lines 12-16), the algorithm identifies the most effective test scenarios. It assigns weights to each transition using the Hybrid Cost Function (C_{hybrid}) defined in Equation (X). Then, Greedy Dijkstra's algorithm is applied to extract the optimal path that maximizes test coverage while minimizing redundancy, guided by the weights w_1 (novelty), w_2(risk), and w_3 (requirements).
Following the Refinement Phase described above, the mathematical formulation of the validated graph is constructed as follows.
The First step is to construct the refined abstract graph G' is the set of states, $A'$is the set of actions, and $\delta'$is the transition function. We convert this to an adjacency matrix.

$$\boldsymbol{G' = (S', A', \delta'), \text{where } S' = \{S_1, \dots, S_n\}, A' = \{a_1, \dots, a_m\}, \delta': S' \times A' \rightarrow S'}$$

That is, if $A[i, j] = 1$, it means there exists a transition from state S_i to state S_j.
Super states, which were already consolidated in the refinement phase (previous step) as single nodes representing sub-sequences (e.g., "deceleration → stop" or "hazard detection → emergency braking"), are treated the same as regular nodes in previous steps. They are included in the state set S'as follows:

$$\boldsymbol{S' = \{S_{\text{Lane Keeping Clear}}, S_{\text{Approaching Intersection}}, S^*_{\text{Safe Stop}}, S^*_{\text{Emergency Stop}}, S_{\text{Collision Risk Active}}, S_{\text{Lane Departure}}\}}$$

When constructing the adjacency matrix A or tensor Min subsequent steps, these super states are included as nodes that can connect to other states. For example:

$$\boldsymbol{S_{\text{Lane Keeping Clear}} \xrightarrow{a_{\text{apply\_brake}}} S^*_{\text{Safe\_Stop}}, S_{\text{Collision Risk Active}} \xrightarrow{a_{\text{emergency\_brake}}} S^*_{\text{Emergency Stop}}}$$

This results in a complete graph that incorporates super states. Our approach is demonstrated using the examples provided above.

4.3 Transitive Closure Calculation

1) Initial Graph Definition and Adjacency Matrix Construction
The refined abstract graph is defined as $\boldsymbol{G' = (S', A', \delta')}$. The state set is indexed as $\boldsymbol{S' = \{S_1, \dots, S_n\}}$, and a binary adjacency matrix $\boldsymbol{A \in \{0, 1\}^{n \times n}}$is constructed as follows [21]:

$$\mathbf{A[i,j]} = \begin{cases} \mathbf{1}, & \mathbf{if\ \exists a \in A'\ s.t.\ \delta'(S_i, a) = S_j,} \\ \mathbf{0}, & \mathbf{otherwise.} \end{cases}$$

To account for self-reachability, the identity matrix I is used to initialize:

$$\mathbf{R^{(0)} = A \vee I}$$

where ∨ denotes the Boolean OR operation.
2): Indirect Transition Expansion (Warshall Iteration)
The reachability relation is extended to include indirect paths through intermediate states, forming a transitive closure. The Warshall algorithm's Boolean recurrence relation is:

$$\mathbf{R^{(k)}[i,j] = R^{(k-1)}[i,j] \vee (R^{(k-1)}[i,k] \wedge R^{(k-1)}[k,j]), k = 1, \dots, n,}$$
$$\mathbf{where\ \wedge\ denotes\ the\ AND\ operation.}$$

The final transitive closure matrix is:

$$\mathbf{R = R^{(n)}}$$

This is equivalent to the closed-form expression:

$$\mathbf{R = I \vee A \vee A^2 \vee \cdots \vee A^{n-1}}$$

The matrices below represent the process of calculating R.

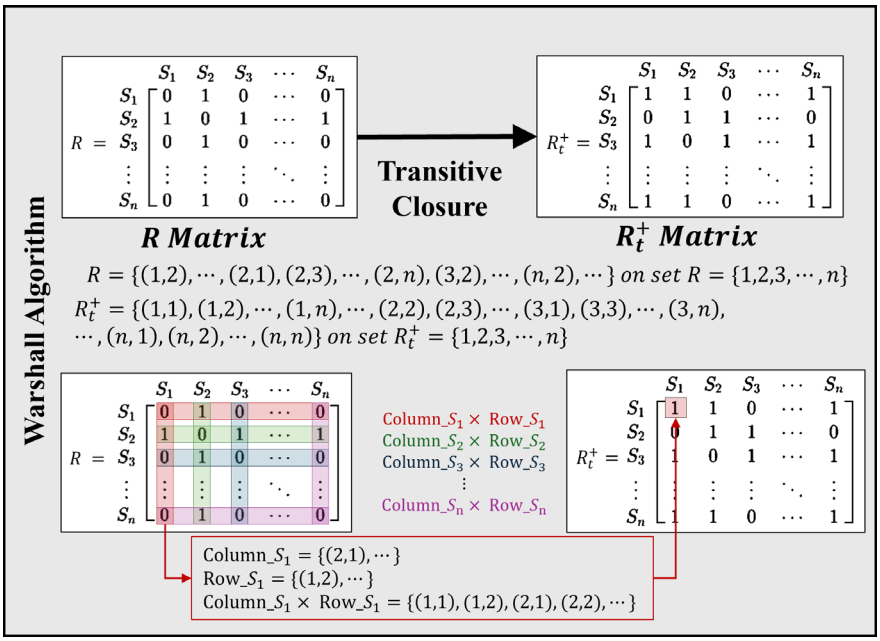


Fig. 6. Warshall's Algorithm with Transitive Closure

This matrix 'A' is an adjacency matrix for a directed graph with six states, where each row and column represents a state, and $\boldsymbol{A[i,j] = 1}$ indicates a transition from state $\boldsymbol{S_i}$to state $\boldsymbol{S_j}$. The adjacency matrix entries indicate specific transitions between states. Then, we will calculate $\boldsymbol{A^2, A^3, A^4, A^5}$step-by-step using an adjacency matrix Boolean multiplication (OR-AND).

$$\boldsymbol{A}=\begin{bmatrix}0&1&1&0&1&1\\0&0&1&0&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}\boldsymbol{A^2}=\begin{bmatrix}0&0&1&1&0&1\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}\boldsymbol{A^3}=\begin{bmatrix}0&0&0&1&0&1\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}$$

$$\mathbf{A^4}=\begin{bmatrix}0&0&0&1&0&1\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}\mathbf{A^5}=\begin{bmatrix}0&0&0&1&0&1\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}\mathbf{A^6}=\begin{bmatrix}0&0&0&1&0&1\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\end{bmatrix}$$

The adjacency matrix A indicates specific state transitions in the system. Specifically, $A[1,3]=1$ signifies that a transition from the "Lane Keeping" state to the "Safe Stop" state is possible. Similarly, $A[5,4]=1$ indicates that a transition from the "Collision Risk" state to the "Emergency Stop" state is feasible. These entries highlight key operational pathways within the system's abstract graph.

$$\mathbf{R}=\mathbf{I}\vee\mathbf{A}\vee\mathbf{A^2}\vee\mathbf{A^3}\vee\mathbf{A^4}\vee\mathbf{A^5}$$

As we see above, the cumulative transition closure is defined as:

$$\mathbf{R}=\begin{bmatrix}1&1&1&1&1&1\\0&1&1&1&0&0\\0&0&1&1&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}$$

The cumulative transition closure matrix R provides insights into state reachability. From state S_1, all states (1 to 6) can be reached, indicating full connectivity. Starting from S_2, only states $\{S_2, S_3, S_4\}$are accessible, excluding hazard or lane departure states. For state S_3, only $\{S_3, S_4\}$ are reachable. State S_4 can only transition to the stop state. Finally, states $S_5$and S_6 are limited to reaching themselves, as the current model includes no further transitions or self-loops for these states.

Integration of Cost Function for Optimal Path Search: Dijkstra's algorithm is a deterministic shortest-path algorithm that finds the path with the minimum total weight between nodes in a weighted graph [22]. It operates by iteratively selecting the node with the smallest cumulative weight, updating the weights of neighboring nodes, and ensuring no negative weights are present. While this deterministic approach may not fully capture the complex, dynamic nature of RL environments, this study transforms this limitation into an advantage. By maintaining the simplicity and computational efficiency of deterministic algorithms, we design semantic cost weights tailored for verification, leveraging Dijkstra's algorithm as an effective tool for test path exploration.

The core objective of this study is not merely to find the "fastest or shortest path." Instead, we redefine the concept of an "optimal path" to achieve "maximum coverage with minimal test cases." Specifically, the proposed algorithm aims to identify the most novel, risky, and critical

paths for validation, prioritizing those with the highest testing value over the shortest or fastest routes.

Weighted value function for RL software validation: This study proposes a hybrid weight function to optimize test path selection, defined as:

$$\boldsymbol{C}_{Integration}(\boldsymbol{s},\boldsymbol{a}) = \boldsymbol{w_1}\cdot\boldsymbol{C}_{\text{novel}}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{w_2}\cdot\boldsymbol{C}_{\text{risk}}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{w_3}\cdot\boldsymbol{C}_{\text{req}}(\boldsymbol{s},\boldsymbol{a})$$

Although mathematically a linear combination, this function serves as a powerful tool for prioritizing test cases based on validation objectives, beyond merely measuring path length. Each weight $\boldsymbol{w_i}$ is adjustable based on the system's functional goals, enabling flexible alignment with specific domain requirements. The weight configuration varies depending on the validation objective. The following **Table 6** outlines key domains and their corresponding weight adjustment strategies:

Table 6. Domain Specific Weight Adjustment

Validation Objective	Key Focus	Weight Adjustment Direction
Safety-Critical Systems	Collision risk, malfunctions, braking failures	Increase w_2(risk), decrease w_1(novelty)
Exploration Testing	New states, rarely visited scenarios	Increase w_1(novelty)
Requirements Validation	Specific function verification, performance thresholds	Increase w_3(requirements)
Adaptive/Autonomous Systems	Balancing stability and adaptability	Balance w_1, w_2, w_3

For instance, in safety-critical systems like autonomous driving, the risk-based weight $\boldsymbol{C}_{\text{risk}}$ should have a higher weight ($\boldsymbol{w_2}$) to prioritize safety-related transitions. Conversely, for exploration-focused testing, the novelty weight $\boldsymbol{C}_{\text{novel}}$ is emphasized with a higher $\boldsymbol{w_1}$to target under-explored scenarios. These weights are tuned based on the experimenter's or domain expert's validation strategy and risk tolerance.

The integration of the cost-weight function proposed in this study offers several key advantages. First, its domain adaptability allows it to be easily tailored to various fields, such as healthcare, autonomous driving, or finance, ensuring relevance across diverse applications. Second, it offers quantitative control, where adjusting a single weight enables precise changes to the testing strategy, allowing focused prioritization of validation objectives. Third, it facilitates integration of data and expertise by combining data-driven visit frequency ($\boldsymbol{V}(\boldsymbol{s},\boldsymbol{a})$) with expert-defined requirement importance ($\boldsymbol{C}_{\text{req}}$), effectively leveraging both training data and domain knowledge. In summary, this hybrid weight function enhances the efficiency and reliability of DRL system verification by optimizing test path selection to achieve maximum coverage with minimal test cases, aligning closely with specified validation goals.

5. Conclusion

Until now, AI researchers have been focused on training reinforcement learning models but haven't yet considered validating them. Therefore, we propose a formal validation mechanism for RL–based software, designed to address the scalability and reliability challenges inherent in complex state–action systems. Our mechanism reduces all possible test-scenario paths by integrating Warshall's dynamic programming algorithm and Dijkstra's optimization. By systematically analyzing the transitive closure of the abstracted state–action graph, the

approach checks whether every goal state is reachable from its initial configuration, thereby detecting logical inconsistencies and requirement gaps early in validation.

Unlike existing heuristic or simulation-based testing approaches, the proposed mechanism performs structural verification before test execution, thereby ensuring the logical soundness of the model. Through this pre-verification, redundant or unreachable paths are eliminated, reducing unnecessary test cases and increasing testing efficiency. The subsequent optimization process, based on deterministic path search, identifies the minimal yet sufficient set of test cases required to achieve maximum coverage. This balance between formal validation and practical optimization allows the validation process to be both mathematically rigorous and computationally efficient. It offers a scalable, domain-independent methodology that can be adapted to various safety-critical applications, such as autonomous vehicles, robotics, and intelligent control systems.

Future work will extend this mechanism to dynamic and stochastic environments, enabling the validation of RL systems that must operate under uncertainty and continuous adaptation.

Acknowledgement

This research was conducted with the support of the Korea Creative Content Agency (Project Name: AI-Based Interactive Multimodal Interactive Storytelling 3D Scene Authoring Technology Development, Project Number: RS-2023-00227917, Contribution Rate: 100%) and the Korea Research Foundation's four, Brain Korea 21 (Project Name: Ultra-Distributed Autonomous Computing Service Technology Research Team, Project Number: 202003520005).

References

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Processing Magazine*, vol.34, no.6, pp.26-38, Nov. 2017. Article(CrossRefLink)

[2] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of Real-World Reinforcement Learning," in *Proc. of the Reinforcement Learning for Real Life (RL4RealLife) Workshop in the 36th International Conference on Machine Learning*, pp.5080-5090, 2019. Article(CrossRefLink)

[3] M. Felderer, and R. Ramler, "Quality Assurance for AI-Based Systems: Overview and Challenges (Introduction to Interactive Session)," in *Proc. of 13th International Conference on Software Quality: Future Perspectives on Software Engineering Quality (SWQD 2021)*, Lecture Notes in Business Information Processing, vol.404, pp.33-42, 2021. Article(CrossRefLink)

[4] A. Aleti, "Software Testing of Generative AI Systems: Challenges and Opportunities," in *Proc. of 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp.4-14, Melbourne, Australia, 2023. Article(CrossRefLink)

[5] A. Gupta and I. Hwang, "Safety Verification of Model Based Reinforcement Learning Controllers," *arXiv preprint arXiv:2010.10740*, 2020. Article(CrossRefLink)

[6] A. J. Singh, and A. Easwaran, "PAS: Probably Approximate Safety Verification of Reinforcement Learning Policy Using Scenario Optimization," in *Proc. of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS '24)*, pp.1745-1753, Auckland, New Zealand, 2024. Article(CrossRefLink)

[7] J. D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software*, vol.10, no.2, pp.14-32, Mar. 1993. Article(CrossRefLink)

[8] Y. Kim, and C. R. Carlson, "Scenario based integration testing for object-oriented software development," in *Proc. of Eighth Asian Test Symposium (ATS'99)*, pp.283-288, Shanghai, China, 1999. Article(CrossRefLink)

[9] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, John Wiley & Sons, 1994. Article(CrossRefLink)
[10] G. H. Mealy, "A Method for Synthesizing Sequential Circuits," *The Bell System Technical Journal*, vol.34, no.5, pp.1045-1079, Sep. 1955. Article(CrossRefLink)
[11] R. Valenzano and F. Xie, "On the Completeness of Best-First Search Variants That Use Random Exploration," in *Proc. of the AAAI Conference on Artificial Intelligence*, vol.30, no.1, 2016. Article(CrossRefLink)
[12] E. F. Moore, "Gedanken-experiments on Sequential Machines," *Automata Studies*, Annals of Mathematical Studies, vol.34, pp.129-153, Princeton University Press, 1956. Article(CrossRefLink)
[13] R. Bellman, "The Theory of Dynamic Programming," *Bulletin of the American Mathematical Society*, vol.60, no.6, pp.503-515, 1954. Article(CrossRefLink)
[14] M. Landers, A. Doryab, "Deep Reinforcement Learning Verification: A Survey," *ACM Computing Surveys*, vol.55, no.14s, pp.1-31, Dec. 2023. Article(CrossRefLink)
[15] V. Mnih, et al., "Human-level control through deep reinforcement learning," *Nature*, vol.518, no.7540, pp.529-533, Feb. 2015. Article(CrossRefLink)
[16] A. Abdurazik, "Generating test cases from UML specifications," MS thesis., George Mason University, May 1999. Article(CrossRefLink)
[17] D. Peled, "Partial-Order Reduction," *Handbook of Model Checking*, pp.173-190, Springer, 2018. Article(CrossRefLink)
[18] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol.8, no.3, pp.231-274, Jun. 1987. Article(CrossRefLink)
[19] A. H. Khan, I. Porres, "Consistency of UML class, object and statechart diagrams using ontology reasoners," *Journal of Visual Languages & Computing*, vol.26, pp.42-65, Feb. 2015. Article(CrossRefLink)
[20] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM (JACM)*, vol.9, no.1, pp.11-12, Jan. 1962. Article(CrossRefLink)
[21] F. Harary, "The Determinant of the Adjacency Matrix of a Graph," *SIAM Review*, vol.4, no.3, pp.202-210, Jul.1962. Article(CrossRefLink)
[22] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Edsger Wybe Dijkstra: His Life,Work, and Legacy*, pp.287-290, Jul. 2022. Article(CrossRefLink)

JANGHWAN KIM received the B.S. degree in computer science from Idaho State University, Pocatello, ID, USA, in 2019, and the M.S. degree from the Department of Software and Communications Engineering, Hongik University, Seoul, South Korea, in 2022. He completed his doctoral coursework (Ph.D. candidate) in the Department of Software and Communications Engineering at Hongik University in 2026. He is currently an Adjunct Professor with the Department of Software Convergence, Hongik University. His research interests include AI software validation, reinforcement learning-based software validation, software visualization, and requirements engineering.

R. YOUNG CHUL KIM received the Ph.D. degree in software engineering from the Department of Computer Science, Illinois Institute of Technology (IIT), USA, in 2000. He worked at LG Research Center, South Korea, in 2001. He is currently a Professor with Hongik University. His research interests include test maturity model, model-based testing, metamodeling, software process model, and software visualization.