Empirical Software Engineering (2021) 26: 52
https://doi.org/10.1007/s10664-021-09941-z

Testing self-healing cyber-physical systems under uncertainty with reinforcement learning: an empirical study

Tao Ma[1,2] • Shaukat Ali[1] • Tao Yue[1,3]

Accepted: 15 January 2021/ Published online: 1 April 2021

Abstract

Self-healing is becoming an essential feature of Cyber-Physical Systems (CPSs). CPSs with this feature are named Self-Healing CPSs (SH-CPSs). SH-CPSs detect and recover from errors caused by hardware or software faults at runtime and handle uncertainties arising from their interactions with environments. Therefore, it is critical to test if SH-CPSs can still behave as expected under uncertainties. By testing an SH-CPS in various conditions and learning from testing results, reinforcement learning algorithms can gradually optimize their testing policies and apply the policies to detect failures, i.e., cases that the SH-CPS fails to behave as expected. However, there is insufficient evidence to know which reinforcement learning algorithms perform the best in terms of testing SH-CPSs behaviors including their self-healing behaviors under uncertainties. To this end, we conducted an empirical study to evaluate the performance of 14 combinations of reinforcement learning algorithms, with two value function learning based methods for operation invocations and seven policy optimization based algorithms for introducing uncertainties. Experimental results reveal that the 14 combinations of the algorithms achieved similar coverage of system states and transitions, and the combination of Q-learning and Uncertainty Policy Optimization (UPO) detected the most failures among the 14 combinations. On average, the Q-Learning and UPO combination managed to discover two times more failures than the others. Meanwhile, the combination took 52% less time to find a failure. Regarding scalability, the time and space costs of the value function learning based methods grow, as the number of states and transitions of the system under test increases. In contrast, increasing the system's complexity has little impact on policy optimization based algorithms.

Keywords Cyber-physical systems · Uncertainty · Self-healing · Model execution · Reinforcement learning · Empirical evaluation

Guest Editor: Hélène Waeselynck

✉ Tao Yue
taoyue@ieee.org

Extended author information available on the last page of the article

1 Introduction

As an essential feature of Cyber-Physical Systems (CPSs), self-healing enables a CPS to autonomously detect and recover from errors caused by software or hardware faults at runtime. We refer to this kind of CPSs as Self-Healing CPSs (SH-CPSs). Besides recovery, SH-CPSs have to address various uncertainties, which mean uncertain values that may affect behaviors of an SH-CPS during execution, including measurement errors from sensors and actuation deviations from actuators. In reality, uncertainties are uncontrollable and exact values of errors are unknown for a given interaction between an SH-CPS and its environment. To assess the reliability of SH-CPSs, we would like to test if an SH-CPS can still behave as expected under uncertainties, with the range of each uncertainty given. As SH-CPSs have two kinds of behaviors (i.e., functional behaviors for fulfilling business requirements and self-healing behaviors for handling faults (Ma et al. 2019a)) both affected by uncertainties, we aim to test both kinds of behaviors of SH-CPSs under uncertainties.

To solve the testing problem, previously, we proposed a fragility-oriented approach (Ma et al. 2019b). In this approach, we evaluate how likely the system will fail in a given state, defined as fragility, and use the fragility as a heuristic to find the optimal testing policies for detecting failures, i.e., unexpected behaviors. Here, we need to find two policies. The first policy is used to decide how to exercise the SH-CPS by invoking its testing interfaces. Meanwhile, another policy is used to determine the value of each uncertainty that affects a measurement or actuation when an SH-CPS uses a sensor or actuator to monitor or change its environment. The value is then passed to simulators of sensors or actuators to replicate the uncertainty's effect. In our previous work (Ma et al. 2019b), reinforcement learning has demonstrated its effectiveness in learning these two policies. Compared with random testing and a coverage-oriented testing approach, the fragility-oriented approach with reinforcement learning discovered significantly more failures. However, as several reinforcement learning algorithms are available (Arulkumaran et al. 2017), there is no sufficient evidence showing which reinforcement learning algorithms are the best to be used for testing SH-CPS under uncertainties.

To this end, we conducted this empirical study, in which the performance of 14 combinations of various reinforcement learning algorithms was evaluated together with the fragility-oriented approach for testing six SH-CPSs. As aforementioned, to detect failures, the algorithms need to learn the optimal policy for invoking testing interfaces (task 1) and learn the best strategy to choose uncertainty values (task 2). As these two tasks are different, we applied two sets of algorithms to perform them. Specifically, we applied two value function learning based algorithms, Action-Reward-State-Action (SARSA) (Sutton and Barto 1998) and Q-learning (Sutton and Barto 1998), for finding the policy of invoking testing interfaces, and seven policy optimization based algorithms, Asynchronous Advantage Actor-Critic (A3C) (Mnih et al. 2016), Actor-Critic method with Experience Replay (ACER) (Wang et al. 2016), Proximal Policy Optimization (PPO) (Schulman et al. 2017), Trust Region Policy Optimization (TRPO) (Schulman et al. 2015), Actor-Critic method using Kronecker-factored Trust Region (ACKTR) (Wu et al. 2017), Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. 2015), and Uncertainty Policy Optimization (UPO) (Ma et al. 2019b), for learning the policy of selecting values for uncertainties.

Results of our empirical study reveal that Q-learning + UPO was the optimal combination that discovered the most failures in the six SH-CPSs under uncertainties. On average, the combination found two times more failures and took 52% less time to find a failure than the others. Regarding the scalability of the applied algorithms, as the numbers of states and transitions of the systems under test increased, the time and space costs of the value function learning based algorithms (SARSA and Q-learning) grew as well, as they had to save values of each state and transition and choose the optimal action based on the values. In contrast, the policy optimization based algorithms were rarely affected by varying complexities of the systems, as they used artificial neural networks to select actions and estimate the values of states and transitions.

The remainder of this paper is organized as follows. Section 2 provides background information, followed by the experiment design in Section 3. Section 4 and Section 5 present the experiment execution and results, with a discussion about the results and alternative approaches given in Section 6. Section 7 analyzes threats to validity. After a discussion about related work in Section 8, Section 9 concludes the paper.

2 Background

This section briefly introduces the test model used to capture components, expected behaviors, and uncertainties of the SH-CPS under test in Section 2.1. Section 2.2 explains how to test the SH-CPS against a test model. The problem is formulated in Section 2.3, and Section 2.4 shows the reinforcement learning algorithms that can be used to solve the testing problem. In this section, key concepts related to testing and reinforcement learning are italicized.

2.1 Uncertainty-Wise Executable Test Model

To test SH-CPSs under uncertainty, in our previous work (Ma et al. 2019a), we have proposed the *Executable Model-Based Testing approach* (*EMBT*). In this approach, an *executable test model* is used to capture components, uncertainties, and expected behaviors of an SH-CPS under test. This section will use an example of an autonomous Copter system to introduce the *test model*, a simplified which is given in Fig. 1.

An *executable test model* consists of a collection of UML[1] class diagrams and state machines. The class diagrams capture components of the SH-CPS under test as UML classes. The components' state variables that are accessible via testing interfaces are specified as properties of the classes and the testing interfaces for controlling or monitoring the components are defined as operations. For example, the Copter has a NavigationUnit and a GPS, and they are specified as UML classes (Fig. 1a). The properties of the NavigationUnit capture its state variables, like "velocity" and "height", that can be queried by the testing interface status(). Besides status(), the NavigationUnit provides several interfaces used to control the flight, including throttle(), pitch(), arm(), and land(). They are all specified as operations of the NavigationUnit. When an SH-CPS uses sensors and actuators to monitor and change its environment, the measurement and actuation performed by the sensors and actuators are affected by uncertainties. Such uncertainties are specified as stereotyped[2] properties, with their ranges specified as stereotype attributes. For instance, the measurement error of "velocity"

[1] UML: Unified Modeling Language (https://www.omg.org/spec/UML/)

[2] Stereotype: it is an extension mechanism provided by UML. We defined a set of stereotypes, also called UML profiles, to extend UML class diagram and state machine to specify components, uncertainties, and expected behaviors of the SH-CPS under test.

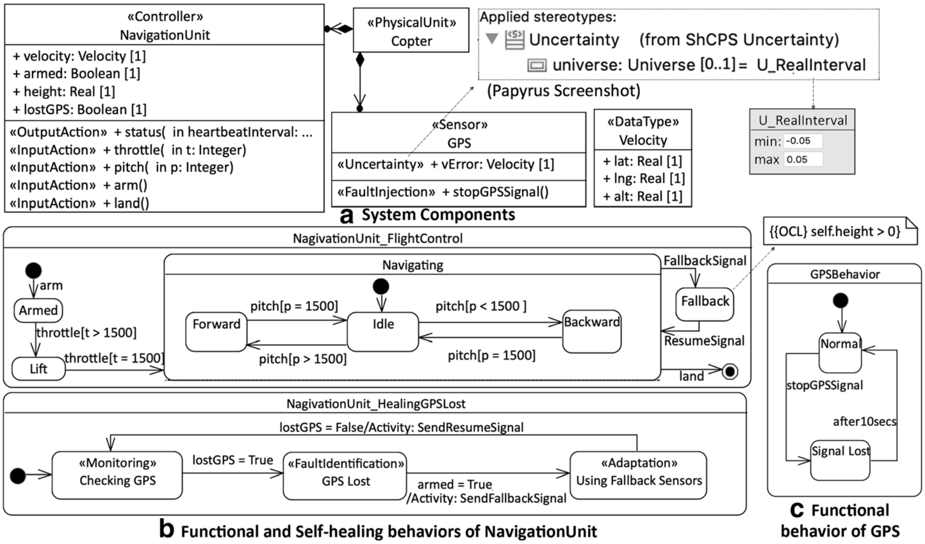

Fig. 1 Simplified Test Model of a Copter System. (**a**) System Components, (**b**) Functional and Self-healing behaviors of Navigation Unit, (**c**) Functional behavior of GPS

measured by GPS is an uncertainty. It is specified as a property of the GPS class, “vError”, and its range is specified by the stereotype attributes: “min” and “max”. Note that as the probability distributions of the uncertainty may not be known, our testing approach does not test SH-CPSs with uncertainties sampled from distributions. Instead, it applies effective algorithms to find the values of uncertainties that can prohibit a system from behaving normally. Therefore, only the valid range of each uncertainty is defined in the test model.

Expected behaviors of each component are specified as UML state machines. A state in a state machine is defined together with a state invariant, which is an OCL[3] constraint of state variables. By evaluating the invariant, with the variables’ values accessed from the system under test, we can check if the invariant is satisfied when a component is supposed to be in a given state. For example, Fig. 1b presents two expected behaviors of the NavigationUnit. The first behavior specifies how the NavigationUnit controls a flight in response to invocations of its operations. For example, the Copter starts to take off when throttle() is invoked with its parameter “t” above 1500. Normally, the NavigationUnit employs the GPS to monitor the flight. When the GPS loses its signal and fails to measure the Copter’s position and velocity, a self-healing behavior (i.e., an adaptive control algorithm) will detect the incorrect measurement, identify the cause (i.e., GPS fault) and switch to other sensors[4] until the GPS outage is over. During this period, the self-healing behavior controls the Copter’s movement properly to avoid it crashing on the ground. The second state machine in Fig. 1b specifies this self-healing behavior. The state invariant of “Fallback” requires that the Copter should be above the ground, i.e., “height > 0”, when the behavior takes effect.

Because the self-healing behavior is internal, it is controlled by the NavigationUnit, instead of external instructions. Consequently, it needs a fault injection operation (e.g.,

[3] OCL: Object Constraint Language. https://www.omg.org/spec/OCL/

[4] Other sensors include barometer, accelerometer and gyroscope. Limited by the space, they are not shown in Fig. 1.

stopGPSSignal() defined in the GPS class) to trigger such behavior. We also use UML state machines to specify when a fault can be injected and how it will affect the state of a corresponding component, with the stereotypes of «Fault» and «Error» provided by our modeling framework (Ma et al. 2019b). For instance, Fig. 1c presents the behavior of GPS. Initially, GPS is in the "Normal" state. Then, stopGPSSignal() can be invoked, and it will trigger GPS switching to the "Signal Lost" state, in which the GPS will stop measuring position and velocity to mimic the fault of losing signal from GPS. After 10 s, GPS will switch back to the "Normal" state and start measuring again. The state machine tells us that this is a transient fault. In contrast, a permanent fault will keep a component in an error state. Based on the UML state machines, an algorithm can inject the fault in various conditions and learn when a fault should be injected to reveal an unexpected behavior.

In summary, a test model captures 1) components, 2) properties, operations, and expected behaviors of each component, and 3) uncertainties that affect the behaviors and ranges of the uncertainties, for an SH-CPS. We aim to test if the SH-CPS behaves consistently with the test model under uncertainties. Note that the purpose of this testing is to detect failures of SH-CPSs under uncertainties. A failure should be differentiated from a fault that is to be healed by self-healing behaviors. The failure that is to be observed by testing is an unexpected behavior, representing a case where an SH-CPS fails to behave consistently with its *test model*. In contrast, the fault that is to be handled by self-healing behaviors should have been identified at design time, and self-healing behaviors have been implemented to detect errors, identify the faults causing the errors, and apply proper adaptations to recover from the errors. For example, the self-healing behavior of the NavigationUnit is designed to detect incorrect GPS measurement (error), caused by the GPS signal loss (fault). If the system fails to detect the error during testing, a failure (unexpected behavior) can be observed.

2.2 Uncertainty-Wise Executable Model-Based Testing

To efficiently test SH-CPSs, we proposed an executable model-based testing approach. In this approach, an SH-CPS is tested against a test model, by executing the system and the model together, sending them the same testing stimuli (e.g., *operation invocation*s) and comparing their consequent states. To realize the approach, we have developed a testing framework. In this subsection, we briefly introduce the theoretical foundations of the executable model-based testing approach in Section 2.2.1 and then present the testing framework in Section 2.2.2. More details about the theoretical foundations and implementation can be found in our previous work (Ma et al. 2019a).

2.2.1 Theoretical Foundations

There are three theoretical foundations underlying this executable model-based testing approach.

First, the standard of Semantics of a Foundational Subset for Executable UML Models (fUML) (OMG 2016), Precise Semantics of UML State Machines (PSSM) (OMG 2017), and our extensions (Ma et al. 2019a) provide precise execution semantics of UML model elements that are used to specify a test model. The semantics enables the test model to be executed in a deterministic manner.

Second, the Object Constraint Language (OCL) (OMG 2006) standard gives us a standard way to specify constraints that an SH-CPS has to satisfy during execution. As explained in the previous section, each state in a test model is defined together with a state invariant, i.e., a constraint on the values of state variables in OCL. By evaluating the invariant with actual values of the state variables obtained from the system under test, we can rigorously check if the system behaves as expected in a given state.

Third, the Functional Mockup Interface (FMI) standard has defined a way to co-execute hybrid models. Based on the standard, we have devised a co-execution algorithm and implemented a testing framework (Ma et al. 2019a) to enable a test model to be executed together with an SH-CPS, even though the test model and components of the SH-CPS are implemented with diverse modeling paradigms.

These three theoretical foundations enable us to co-execute the test model and SH-CPS in a deterministic manner, and also allow us to rigorously check if the system behave consistently with the model.

2.2.2 Implementation (TM-Executor)

To realize the executable model-based testing, we have developed a testing framework, TM-Executor. Figure 2 presents an overview of the framework. As shown in the figure, a *test model* is executed by an Execution Engine, together with the SH-CPS under test. To drive the execution under uncertainties, a Test Driver has to invoke operations on the system and the model to control their behaviors. Meanwhile, an Uncertainty Introducer needs to introduce uncertainties in the environment to replicate the effects of measurement errors and actuation deviations via simulators of sensors and actuators. The two parallel processes determine how an SH-CPS is tested under uncertainties.

For *operation invocations*, the Test Driver takes the current active state and its outgoing transitions as input and outputs an *operation invocation* that is to be performed by the Execution Engine to make the SH-CPS and test model switch to a consequent state. The *operation invocation* is defined as follows:

Definition 1. An *operation invocation* **is a combination of an operation and its input parameter values that are used to call the operation and trigger a transition defined in a** *test model***.** For instance, when the active states of the NavigationUnit and GPS are ("Idle",

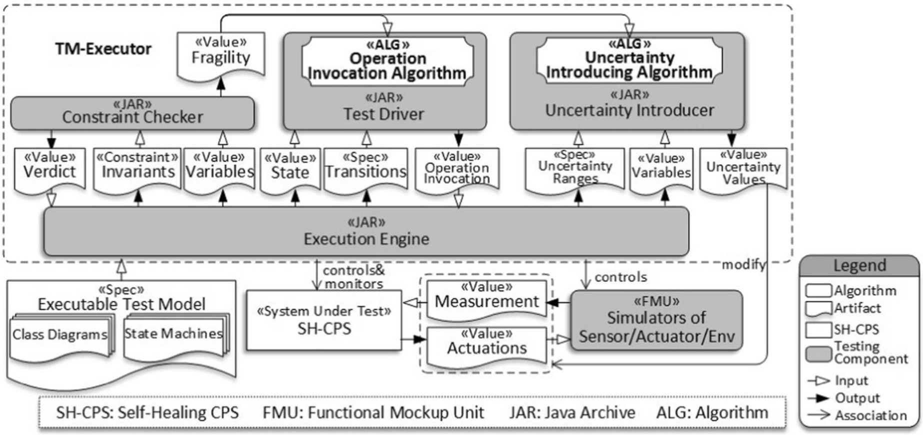

Fig. 2 Overview of the TM-Executor Testing Framework

"Checking GPS", "Normal") as shown in Fig. 1, the Test Driver needs to choose either to invoke pitch() to let the NavigationUnit switch to state "Forward" or "Backward", or to call stopGPSSignal() to inject a GPS fault. When an operation is selected, and an *operation invocation* is generated by the Test Driver, the Execution Engine will perform the invocation to trigger an outgoing transition in the test model. Meanwhile, the Execution Engine invokes the testing interface represented by the operation with the same input parameter values, to make the system enter the target state of the outgoing transition as well. To check if the consequent states of the SH-CPS and *test model* are the same, the Execution Engine obtains state variables' values from the SH-CPS via testing interfaces, and passes the values to a Constraint Checker[5] to evaluate the invariant of the target state. If the invariant is not satisfied, it means that the SH-CPS failed to behave consistently with the *test model*, and thus a failure is revealed. Otherwise, the Test Driver will keep on generating *operation invocation*s to drive the execution until a terminal state is reached.

On the other hand, the Uncertainty Introducer needs to select an *uncertainty value* for each uncertainty and each measurement or actuation that the uncertainty may take effect. The definition of *uncertainty value* is given below:

> Definition 2. An *uncertainty value* is an exact value of a measurement error or actuation deviation.

The *uncertainty value* is used to modify measurements performed by the sensors and actions performed by actuators to simulate the effect of uncertainties. For example, an *uncertainty value* is chosen for the measurement error of GPS, "vError" (shown in Fig. 1a), for each velocity measured by the GPS. By adding the measurement error ("vError") to the true value of the velocity, derived from a simulation model, we can replicate the effect of the uncertainty, and test if the system will violate any invariants with the selected measurement errors.

With the help of TM-Executor, we can execute an SH-CPS together with its test model, and check if they are behaving consistently under uncertainties, given the ranges of uncertainties. The remaining problem is how to efficiently explore various sequences of *operation invocations* and *uncertainty values* to find the ones that can reveal a failure. In the next section, we will see how reinforcement learning can be used to solve this problem.

2.3 Problem Formulation

As explained in the previous section, testing an SH-CPS under uncertainties involves two parallel processes: 1) invoking operations on the system to explore its behaviors and 2) introducing uncertainties in the environment to simulate the effects of measurement errors and actuation deviations. The two processes are independent from each other, since in the real-world the uncertainties keep changing, independent of operation invocations performed on the system.

To find a sequence of *operation invocation*s, S_i, and a sequence of *uncertainty value*s, S_u, that can work together to make an SH-CPS violate an invariant, we can either find them concurrently or find them sequentially, i.e., find a sequence of *operation invocation*s first and then find uncertainty values that make the SH-CPS violate an invariant during handling the

[5] DresdenOCL (https://github.com/dresden-ocl/standalone) is used in TM-Executor to evaluate OCL constraints.

*operation invocation*s. In case they are to be found concurrently, S_i and S_u are to be found from $O * U$ candidate solutions, where O is the number of all possible sequences of *operation invocation*s and U is the number of all possible sequences of *uncertainty value*s. Alternatively, if we find them sequentially and S_i is found as the n$^{\text{th}}$ best solution, with uncertainty values only uniformly sampled from their ranges, then S_u can be found with the top n best sequences of *operation invocation*s. Consequently, S_i and S_u only need to be found from $O + n * U$ candidate solutions. Under the assumption that S_i can still lead to a high chance of detecting a failure when uncertainty values are uniformly sampled, the "n" will be small, and thus $O + n * U$ will be much less than $O * U$. Therefore, we chose to solve the testing problem by sequentially resolving two tasks:

> Task 1 Given a *test model*, find the optimal sequence of *operation invocation*s to maximize the chance of detecting failures, with uncertainties uniformly sampled from their ranges
>
> Task 2 Given a *test model* and a sequence of *operation invocation*s, find the sequence of *uncertainty value*s that makes the SH-CPS under test violate an invariant during handling the *operation invocation*s

Using the terms from reinforcement learning, the two tasks can be rephrased as finding the optimal *policy* of choosing *action*s (i.e., *operation invocation*s or *uncertainty value*s) for an *agent* (i.e., Test Driver or Uncertainty Introducer) to maximize a long-term reward (i.e., *fragility* that indicates the chance to detect a failure). Formally, the *fragility* is defined as follows.

> Definition 3. *Fragility* is defined as a distance that indicates how likely a state invariant is to be violated:

$$F(s_t) = 1 - dis(\neg o_t, V) \tag{1}$$

where o_t is a state invariant, i.e., a constraint on the values of a set of state variables in OCL (Section 2.1), V is the values of state variables, $dis(\neg o_t, V)$ is a distance function that returns a value between 0 and 1 indicating how close the constraint $\neg o_t$ is to be satisfied by V. The distance functions for all types of OCL constraints can be found in (Ali et al. 2013).

Take the Copter shown in Fig. 1 as an example. Assume the Copter is in the "Fallback" state. Its state invariant is "height > *0*", where height is a state variable, representing the distance between the Copter and ground. This invariant requires that height must be larger than zero to avoid crashing on the ground (i.e., "height = 0" means crashing). If height equals to 10, the *fragility* of the Copter equals to: $1 - dis[\neg(height \ \ > 0)] = 1 - \frac{10}{10+1} = 0.09$, according to the distance function given in (Ali et al. 2013). If the height is reduced to 1, then the *fragility* will be increased to 0.5, indicating the Copter is closer to crash on the ground. When height is reduced to zero, and the invariant is violated, the *fragility* is increased to one, its maximum value.

In the context of testing, the purpose of the *agent*s is to discover failures, and thus they are interested in finding *action*s that can lead to a violation of an invariant, i.e., making *fragility* equal to 1, rather than increasing the sum of *fragilities*. For instance, one sequence of *action*s makes an SH-CPS go through three states: s_1, s_2, and s_3, with their *fragilities* being 0.0, 0.0, and 0.9, respectively. Another sequence leads to states $s_1^{'}$, $s_2^{'}$, $s_3^{'}$, $s_4^{'}$, and $s_5^{'}$ with their fragilities being 0.1, 0.2, 0.1, 0.3, and 0.3, respectively. Though the later sequence of *action*s obtains a higher sum of *fragilities*, it is less likely to detect a failure than the first sequence. Therefore, we adapt the objective of reinforcement learning from maximizing cumulative *reward*s to maximizing a future *reward*, i.e., increasing the maximum *fragility* that can be reached in the future, as defined below:

$$J(\theta) = \mathbb{E}_{\pi}[\max_{t\in[1,\infty)}(\gamma^{t}\cdot F(s_t))] \tag{2}$$

where π denotes a policy used to choose *action*s, which can be either *operation invocation*s or *uncertainty value*s; $\mathbb{E}_{\pi}[...]$ means the expected value when π is used to select *action*s; γ is a discount factor, between 0 and 1. It determines the importance of future *fragilities*. If γ equals to 1, the *fragility* that can be reached in the future is considered equally important as the recent ones. On the contrary, if γ is 0, the algorithms will consider only the next *fragility* after taking a selected *action*. Based on the adapted objective, we also need to update two value functions that are broadly used by reinforcement learning algorithms, as discussed below.

> Definition 4. State value function represents the highest discounted *fragility* that can be reached, starting from a given state s^* and thereafter following policy π:

$$V_{\pi}(s^*) = \mathbb{E}_{\pi}[\max_{k\in[0,\infty)}\gamma^{k}\cdot F(s_{t+k})|s_t = s^*, a_{t+k}\sim\pi] \tag{3}$$

where $\mathbb{E}_{\pi}[...]$ denotes the expected value, γ is the discount factor, $F(s_{t+k})$ represents the *fragility*, and π is the policy of selecting *action*s. $a_{t+k}\sim\pi$ means choosing *action*, a_{t+k}, by following the policy π.

> Definition 5. Action value function, also called Q function, specifies the Q value — the highest discounted fragility that can be obtained, when taking *action* a^* in state s^* and then following policy π:

$$Q_{\pi}(s^*, a^*) = \mathbb{E}_{\pi}[\max_{k\in[0,\infty)}\gamma^{k}\cdot F(s_{t+k})|s_t = s^*, a_t = a^*, a_{t+k}\sim\pi] \tag{4}$$

Based on the adapted objective, we can apply reinforcement learning algorithms to address the two tasks sequentially. For Task 1, an algorithm takes the current active state and its outgoing transitions as inputs. As a state only has finite outgoing transitions, the input space of the algorithm is finite and discrete. The output of the algorithm is one of the outgoing transitions. A test data generator, EsOCL (Ali et al. 2013), is used to generate an *operation invocation* (including an operation and valid inputs of the operation) to activate the trigger specified on the transition. Consequently, the algorithm only needs to choose one of the outgoing transitions as output, and thus its output space is also discrete. After a number of episodes,[6] the sequence of *operation invocation*s that leads to the highest *fragility* is chosen to be the optimal one. It is used in Task 2 to find the uncertainty values that can reveal a system failure. For the algorithms

[6] An episode is to execute an SH-CPS from an initial state to a final state once.

used to address Task 2, their inputs are the ranges of *uncertainty value*s and the *state* of the SH-CPS under test, reified as state variables' values of the system, like the velocity and position of the Copter. The output of the algorithm is an *uncertainty value* for each uncertainty. As the state variables' values and *uncertainty value*s are continuous, both input and output spaces of the algorithm are continuous. As the two tasks have different characteristics, they pose different requirements for reinforcement learning algorithms. The following section presents the state-of-the-art algorithms for solving the tasks (Table 1).

2.4 Reinforcement learning algorithms

In general, reinforcement learning algorithms can be classified into value function learning based approaches and policy optimization based approaches (Arulkumaran et al. 2017). Based on these two categories and more detailed subcategories proposed in literature reviews (Duan et al. 2016; Arulkumaran et al. 2017; Kiumarsi et al. 2017) we selected a benchmark reinforcement learning algorithm for each subcategory, summarized in Tables 2 and 1. More details are given in the following subsections.

2.4.1 Value function learning methods

The essence of value function based reinforcement learning algorithms is temporal difference learning, that is, to reduce the difference between the *Q value* estimated at time step *t* and the *Q value* estimated for the next time step $t+1$. The difference is also called *temporal difference error*. When the error is reduced to zero for all *state-action* pairs, the Q function is learned. By selecting the *action* with the highest *Q value*, we can obtain the optimal policy.

Table 1 Policy Optimization Based Reinforcement Learning Algorithms*

Algorithm	Policy Evaluation	Gradient Used to Update Policy	Method to Reuse Samples
A3C	$A_\pi(s_t, a_t)$	$\nabla_\theta \log \pi_\theta(a_t \vert s_t) \cdot A_\pi(s_t, a_t)$	N/A
ACKTR		$\nabla_\theta^{K-FAC} \log \pi_\theta(a_t \vert s_t) \cdot A_\pi(s_t, a_t)$	
DDPG	$Q_\pi(s_t, a_t)$	$\nabla_\theta \log \pi_\theta(a_t \vert s_t) \cdot Q_\pi(s_t, a_t)$	
TRPO		$\nabla_\theta^{con} \log \pi_\theta(a_t \vert s_t) \cdot Q_\pi(s_t, a_t)$, subject to $D_{KL}(\pi_{old} \vert \pi_\theta) < \delta$	Important sampling
PPO	$A_\pi(s_t, a_t)$	$\nabla_\theta \log \pi_\theta(a_t \vert s_t) \cdot A_\pi^{clip}(s_t, a_t)$	Clipped important sampling
ACER		$\min\{c, w_t\} \cdot \nabla_\theta \log \pi_\theta(a_t \vert s_t) \cdot [Q^{Ret}(s_t, a_t) - V_\pi(s_t)] + Corr_{bias}$, subject to $D_{KL}(\pi_{avg} \vert \pi_\theta) < \delta$	Truncated important sampling with bias correction
UPO	N/A	$\nabla_\theta^{con} \log \pi_\theta(a_t \vert s_t) \cdot Q_{best}(s_t, a_t)$	N/A

* Policy optimization based algorithms maintain a *policy network* (an artificial neural network) to select *action*s. To make the *policy network* converge to the optimal policy, the algorithms 1) collect samples of (*state*, *action*, *reward*) by following the *policy network*, 2) optionally use the samples to evaluate the current policy, and 3) optimize the *policy network* based on the samples or evaluation. $\boldsymbol{A_\pi(s_t, a_t)}$ is the *advantage function*; $\boldsymbol{\pi_\theta(a_t \vert s_t)}$ is a policy controlled by parameters $\boldsymbol{\theta}$; $\boldsymbol{\nabla_\theta}$ is the gradient with respect to the parameters of the policy; $\boldsymbol{\nabla_\theta^{K-FAC}}$ is an approximated *natural gradient* by K-FACC; $\boldsymbol{\nabla_\theta^{con}}$ is another approximated *natural gradient* by conjugate gradient algorithm; $\boldsymbol{D_{KL}(\pi_{old} \vert \pi_\theta)}$ is KL divergence between $\boldsymbol{\pi_{old}}$ and $\boldsymbol{\pi_\theta}$; $\boldsymbol{A_\pi^{clip}}$ is clipped advantage function; $\boldsymbol{w_t}$ is importance weight; $\boldsymbol{Q^{Ret}(s_t, a_t)}$ is a *Q function* estimated by *Retrace*; $\boldsymbol{Corr_{bias}}$ is a bias correction term used by ACER; $\boldsymbol{Q_{best}(s_t, a_t)}$ represents the *Q values* of the optimal *action*s that have been found so far

Table 2 Value Function Learning Based Reinforcement Learning Algorithms*

Algorithm	Exploration Policy	Value Function Learning
SARSA	ε-greedy	$Q_\pi(s_t, a_t) = Q_\pi(s_t, a_t) + \alpha \cdot Err_{\pi,\ t}$
Q-learning		$Q_*(s_t, a_t) = Q_*(s_t, a_t) + \alpha \cdot Err_{*,\ t}$

* Value function learning based reinforcement learning algorithms apply an exploration policy, e.g., ε-greedy, to select *action*s based on their *Q value*s. They try to learn *Q value*s and select *action*s with the highest *Q value*s, thus they do not need to learn an explicit policy for selecting *action*s. $\boldsymbol{Q_\pi(s_t, a_t)}$ is estimated *Q function* for policy π; $\boldsymbol{Q_*(s_t, a_t)}$ **is** estimated *Q function* for the optimal policy; $\boldsymbol{\alpha}$ is a learning rate; ***Err*** is *temporal difference error*

State-Action-Reward-State-Action (SARSA) (Sutton and Barto 1998) SARSA uses the following equation to learn Q_π for policy π.

$$Q_\pi(s_t, a_t) = Q_\pi(s_t, a_t) + \alpha \cdot Err_{\pi,\ t} \tag{5}$$

where Q_π is estimated *Q function* for π; α is a learning rate, which controls the step size of each update; $Err_{\pi,\ t}$ is the *temporal difference error*. Based on the adapted objective of reinforcement learning (Section 2.3), $Err_{\pi,\ t}$ is calculated by $max[F(s_t), \gamma \cdot Q_\pi(s_{t+1}, a_{t+1})] - Q_\pi(s_t, a_t)$. When a sample of (*state*, *action*, *reward*) is obtained, SARSA takes Eq. (5) to update $Q_\pi(s_t, a_t)$. To collect the sample, SARSA applies the ε-greedy policy to select *action*s. That is, with a probability of ε, the policy randomly selects from all possible *action*s, and with a probability of 1- ε, it selects the *action* with the highest Q value. In theory, SARSA can converge to the *Q function* of the optimal policy, as long as all *state-action* pairs are visited an infinite number of times, and the ε-greedy converges in the limit to the greedy policy, i.e., reducing ε to zero (Singh et al. 2000).

Q-learning (Sutton and Barto 1998) Instead of learning the *Q function* of a given policy as SARSA does, Q-learning tries to learn the *Q function* of the optimal policy directly, independent of the policy being followed:

$$Q_*(s_t, a_t) = Q_*(s_t, a_t) + \alpha \cdot Err_{*,\ t} \tag{6}$$

where $Q_*(s_t, a_t)$ represents the estimated *Q function* of the optimal policy, and $Err_{*,\ t}$ is the temporal difference error for Q_*, calculated by $max[F(s_t), \gamma \cdot Q_*(s_{t+1}, a_{t+1})] - Q_*(s_t, a_t)$. In this case, the policy, π, used by Q-learning only determines which *state-action* pair is to be visited, while the *state-action* pair and observed *reward* are used to update Q_*, rather than Q_π. As all pairs of *state-action* continue to be visited, Q-learning will gradually learn the *Q function* and find the corresponding optimal policy (Sutton and Barto 1998).

2.4.2 Policy optimization methods

In contrast to value function based methods, policy-based reinforcement learning algorithms maintain a *policy network* (an Artificial Neural Network (ANN)) to select *action*s. The *policy network* takes the *state* of the *environment* as input and outputs an *action* that is to be performed on the *environment*. By following the *policy network*, the policy-based algorithms collect samples of (*state*, *action*, *reward*), optionally take the samples to evaluate the policy determined by the *policy network*, and then optimize the *policy network* based on the samples or evaluation. The main differences among the policy-based algorithms lay in the method of policy evaluation, optimization, and whether/how samples can be reused for the evaluation.

Asynchronous Advantage Actor-Critic (A3C) (Mnih et al. 2016) In A3C, multiple threats are run in parallel to collect samples of (*state*, *action*, *reward*) by following a threat dependent *policy network*. The samples are used to train a *critic* (another ANN) that estimates an *advantage function* for evaluating the policy. The *advantage function*, $A_\pi(s_t, a_t)$, equals to $Q_\pi(s_t, a_t) - V_\pi(s_t)$. It reveals how good an *action* a_t is to be taken in a given *state* s_t, compared with the average value of all candidate *action*s in *state* s_t. Each threat updates its *policy network* by the gradient of the "goodness" of *action*s with respect to the parameters of its policy, i.e., $\nabla_\theta \log \pi_\theta(a_t | s_t) \cdot A_\pi(s_t, a_t)$. From time to time, the local *policy network*s are synced with a global one, so that the threats can work together to learn the optimal policy.

Actor-Critic method using Kronecker-factored Trust Region (ACKTR) (Wu et al. 2017) Compared with the gradient taken by A3C, a *natural gradient* can give a better direction for improvement. However, computing *natural gradient* is extremely expensive. To reduce the computational complexity, ACKTR proposes to use Kronecker-Factored Approximated Curvature (K-FAC) (Martens and Grosse 2015) to obtain an approximate *natural gradient* and take the approximate gradient to optimize the *policy network* and *critic*.

Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. 2015) From another perspective, DDPG uses *Q function* instead of the *advantage function* to evaluate the goodness of *action*s. It calculates the gradient of *Q function* with respect to the policy's parameters, $\nabla_{a_t} Q(s_t, a_t) \cdot \nabla_\theta \pi_\theta(a_t | s_t)$, and uses the gradient to update the *policy network* to make the network select *action*s with the highest *Q value*.

Trust Region Policy Optimization (TRPO) (Schulman et al. 2015) A shortcoming of aforementioned algorithms is that they need to recollect samples of (*state*, *action*, *reward*) to evaluate the *policy network* after each update. To improve the sample efficiency, *importance sampling* can be used as an off-policy estimator to estimate the *advantage function* or *Q function* of a given policy, using samples collected under other policies:

$$\widehat{A}_{\pi_\theta}(s_t, a_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_{\pi_{\theta_{old}}}(s_t, a_t) \tag{7}$$

$$\widehat{Q}_{\pi_\theta}(s_t, a_t) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} Q_{\pi_{\theta_{old}}}(s_t, a_t) \tag{8}$$

where $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is called *importance weight*. However, if $\pi_\theta(a_t| s_t)$ deviates too much from $\pi_{\theta_{old}}(a_t|s_t)$, importance sampling will have high variance. Imagine that, if $\pi_{\theta_{old}}(a_t|s_t)$ is zero for a pair of (s_t, a_t) and $\pi_\theta(a_t| s_t)$ is greater than zero, the *importance weight* will become infinite. Therefore, to use importance sampling, it is necessary to bound the difference between the two policies. To do so, TRPO adds a *KL divergence* (Pollard 2000) constraint to each policy update, and it transforms the reinforcement learning problem into a constrained optimization problem:

$$\begin{aligned} &\text{maximize} \quad \mathbb{E}_{s_t \sim \rho_{\theta_{old}}, a_t \sim \pi_{\theta_{old}}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} Q_{\theta_{old}}(s_t, a_t)\right] \\ &\text{subject to} \quad \overline{D_{KL}}(\theta_{old}|\theta) \le \delta \end{aligned} \tag{9}$$

where $\rho_{\theta_{old}}$ is the *state* distribution determined by policy $\pi_{\theta_{old}}$; $Q_{\theta_{old}}$ is the *Q function* of policy $\pi_{\theta_{old}}$; $\overline{D_{KL}}(\theta_{old}|\theta)$ is average *KL divergence* between two policies; and δ controls the maximum step size for one policy update. The KL divergence constraint defines a trust region for a policy update. When the constraint is met, TRPO can guarantee a monotonic improvement for the *policy network*. To efficiently solve the constrained optimization problem, TRPO applies the Conjugate Gradient Algorithm (Pascanu and Bengio 2013) to approximately calculate *natural gradient* and follows the direction of the gradient to find the solution of the optimization problem.

Proximal Policy Optimization (PPO) (Schulman et al. 2017) As TRPO is relatively complicated, PPO was proposed to use a clip function as an alternative to the *KL divergence* constraint. The clip function is:

$$clip(w_t, 1-\varepsilon, 1+\varepsilon) = \begin{cases} 1+\varepsilon, & if\ \ 1+\varepsilon \leq w_t \\ w_t, & if\ 1-\varepsilon \leq w_t < 1+\varepsilon \\ 1-\varepsilon, & if\ w_t < 1-\varepsilon \end{cases} \tag{10}$$

where w_t is the *importance weight*, $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. PPO uses the clip function to bound the value of *importance weight* and takes the gradient, $\nabla_\theta \log\pi_\theta(a_t|s_t)\cdot A_\pi^{clip}(s_t,a_t) = \nabla_\theta log\pi_\theta(a_t|s_t)\cdot clip(w_t, 1-\varepsilon, 1+\varepsilon)\cdot A_{\pi_{\theta_{old}}}(s_t,a_t)$, to update the *policy network*. However, the clip function will introduce a bias to the estimation of the *advantage function*, which could lead to a suboptimal policy learned by the algorithm.

Actor-Critic method with Experience Replay (ACER) (Wang et al. 2016) To improve sample efficiency and stabilize the estimation of the value function, ACER was proposed with three techniques. First, it uses *Retrace* (Munos et al. 2016) to estimate the *Q function* of the current policy, using samples collected under past policies. As proven in (Munos et al. 2016), *Retrace* has low variance, and it can converge to the *Q function* of a given policy, using samples collected from any policies.

Second, ACER truncates *importance weight*s and adds a bias correction term to reduce the bias caused by the truncation. Particularly, it takes the following gradient to update its *policy network*:

$$\begin{aligned} g^{acer} = {} & min\{c, w_t\}\cdot\nabla_\theta\, log\,\pi_\theta(a_t|s_t)\cdot\left[Q^{Ret}(s_t,a_t) - V_{\pi_{\theta_{old}}}(s_t)\right] \\ & + \mathbb{E}_{a\sim\pi}\left(\left[\frac{w_t(a)-c}{w_t(a)}\right]_+ \cdot\nabla_\theta\, log\,\pi_\theta(a_t|s_t)\cdot\left[Q_{\pi_{\theta_{old}}}(s_t,a_t) - V_{\pi_{\theta_{old}}}(s_t)\right]\right) \end{aligned} \tag{11}$$

where w_t is the *importance weight*; c is a threshold used to truncate the *importance weight*; $Q^{Ret}(s_t, a_t)$ is a *Q function* estimated by *Retrace*; $V_{\pi_{\theta_{old}}}(s_t)$ and $Q_{\pi_{\theta_{old}}}$ are *state value function* and *Q function* estimated by a *critic*, using samples collected under past policies; $\left[\frac{w_t(a)-c}{w_t(a)}\right]_+$ equals to $\frac{w_t(a)-c}{w_t(a)}$ when $w_t(a)$ is greater than c, and it is zero otherwise. Intuitively, if the *importance weight* is less than the threshold, it means that the policy π_θ does not deviate much from $\pi_{\theta_{old}}$, and *Retrace* can give a relatively accurate estimation. Consequently, $Q^{Ret}(s_t, a_t)$ is taken to calculate the gradient used for a policy update. In contrast, when $w_t > c$, as π_θ and $\pi_{\theta_{old}}$ are too different, the *importance weight* becomes too large and the *Retrace* estimation is unreliable to be used alone. Therefore, the *importance weight* is truncated, and the *Q* value estimated by the *critic*, i.e., $Q_{\pi_{\theta_{old}}}(s_t, a_t)$, is used to compensate for the truncation.

Third, to further stabilize the learning process, ACER adds a KL divergence constraint. Different from TRPO that limits the KL divergence between updated and current policies, ACER maintains an average policy, representing all past policies and constrains an updated policy not deviating too much from the average.

Uncertainty Policy Optimization (UPO) (Ma et al. 2019b) Different from the aforementioned methods, which apply a *critic* to evaluate their policy, UPO directly searches the space of all possible policies to find the optimal one. In UPO, the policy is decomposed into a probability distribution and a *policy network* that outputs statistics of the distribution. For example, if we choose to use the normal distribution, the outputs of the *policy network* will be mean and variance of the distribution. Therefore, given a distribution, the *policy network* determines the policy used to select *action*s. In the beginning, UPO starts with a randomly initialized *policy network*, and it keeps on selecting *action*s by following the policy determined by the network. When UPO observes a sequence of *action*s leading to a higher *fragility*, it calculates the conjugate gradient (Pascanu and Bengio 2013) of the *policy network*, multiplied with Q value, i.e., $\nabla_\theta^{con}\log\pi_\theta(a_t|s_t) \cdot Q_{best}(s_t, a_t)$, where $\nabla_\theta^{con}\log\pi_\theta(a_t|s_t)$ is the conjugate gradient and $Q_{best}(s_t, a_t)$ is the Q *value* of a pair of *state* and *action* that has been observed by performing the sequence of *action*s. Afterward, UPO takes the gradient to update parameters of the *policy network*, so as to increase the selection probability of this sequence of *action*s. UPO continues the search process until reaching the maximum iterations.

3 Experiment Planning

Following the guidelines of conducting and reporting empirical studies (Wohlin et al. 2012) (Jedlitschka et al. 2008) (Kitchenham et al. 2002), we designed and conducted the experiment. Section 3.1 presents our research goals. Sections 3.2 to 3.4 describe the rationale of choosing candidate algorithms, subject systems, and testing tasks performed by the algorithms. Hypotheses and related variables are described in Section 3.5, and Section 3.6 explains the applied statistical tests.

3.1 Goals

The objective of the empirical study is to find the best reinforcement learning algorithms for testing SH-CPSs under uncertainty. More specifically, we would like to identify the optimal algorithm for choosing operation invocations and the optimal algorithm for selecting uncertainty values, such that the two algorithms can work together to discover the most failures, and preferably take the least amount of time. Meanwhile, we would also like to investigate the scalability of the algorithms to assess the feasibility of applying them to test complex SH-CPSs. Consequently, we defined two goals for the empirical study.

> **Goal 1.** In the context of uncertainty-wise executable model-based testing, analyze the effectiveness and efficiency of the reinforcement learning algorithms to determine the optimal algorithms of invoking operations and introducing uncertainties, for discovering failures of SH-CPSs under uncertainty.
> **Goal 2.** In the context of uncertainty-wise executable model-based testing, analyze the scalability of the reinforcement learning algorithms to examine their abilities to be applied for testing complex SH-CPSs.

3.2 Algorithms under Investigation

As explained in Section 2.3, testing SH-CPSs under uncertainty is comprised of two tasks: selecting a sequence of operation invocations and choosing a sequence of uncertainty values. Because the two tasks have different requirements, which cannot be satisfied by all the algorithms introduced in Section 2.4, we chose different sets of reinforcement learning algorithms to address these two tasks.

For operation invocations, an algorithm has to select one of outgoing transitions of the current active state, specified in a test model. As the set of outgoing transitions varies from state to state, the algorithm has to choose a transition from an unfixed set of options. However, for the reinforcement learning algorithms using Artificial Neural Networks (ANN), the set of candidates has to be fixed, as an ANN has to have fixed numbers of inputs and outputs. Hence, we only applied the remaining two algorithms (i.e., Q-learning and SARSA) to solve the first task.

For the task of choosing uncertainty values, an algorithm has to select a value for each uncertainty, whenever an SH-CPS interacts with its environment via sensors or actuators. Assume an SH-CPS is affected by k uncertainties, each uncertainty has n possible values, and the CPS interacts with its environment m times. In total, $n^k \times m$ combinations of uncertainty values need to be selected. As value function learning based algorithms (Section 2.4.1) have to learn the Q value for each combination and find the combination with the highest Q value, it is too computationally expensive for them to handle such a huge number of combinations. In contrast, policy optimization based algorithms (Section 2.4.2) explicitly maintain a policy network to select actions, instead of choosing actions based on their Q values, and thus they can efficiently select an action from a huge set of options. Therefore, we applied the policy optimization methods (i.e., A3C, ACER, PPO, TRPO, ACKTR, DDPG, and UPO) for the second task.

In summary, two reinforcement learning algorithms were used for selecting operation invocations, and seven algorithms were applied to choose uncertainty values. In total, there are 14 combinations of the algorithms, also called 14 testing approaches (denoted as *APP*) in the following. We implemented SARSA, Q-learning, and UPO by ourselves, and the other algorithms were taken from OpenAI Baselines.[7]

3.3 Subject Systems

To evaluate the performance of the algorithms, we employed six SH-CPSs from different domains, with diverse complexities, for the empirical evaluation. Three of them are real-world systems, and the others are from the literature. Section 3.3.1 introduces their functionalities, self-healing behaviors, and associated uncertainties. Section 3.3.2 explains how test models were specified for the six SH-CPSs.

3.3.1 System Description

ArduCopter (AC)[8] is a full-featured, open-source control system for multicopters, helicopters, and other motor vehicles. It can cater to a range of flight requirements, from First Person View

[7] https://github.com/openai/baselines

[8] http://ardupilot.org/copter/

racing, to aerial photography and autonomous cruising. It is equipped with five **self-healing behaviors**. Two of them are rule-based policies used to detect the disconnection between a copter and its radio controller or ground control station, and guide the copter to return and land. Another self-healing behavior is a quantitative model-based method (Venkatasubramanian et al. 2003) for detecting measurement errors caused by a transient GPS fault. When such an error is detected, the behavior will identify the fault and adapt the copter to use other sensors. The other two self-healing behaviors are two control algorithms that are used in the event of high vibration, e.g., strong wind, to stabilize the flight, and used to avoid collision with an intruding aerial vehicle. In total, AC uses four sensors (one GPS, one accelerometer, one gyroscope, and one barometer) and one actuator (a motor) to monitor and control the flight. Table 3 shows the eight types of uncertainty related to these sensors and the actuator. The ranges of the uncertainties are specified in their product specification. Each type of uncertainty affects measurement errors or actuation deviations in three dimensions, i.e., longitude, latitude, and altitude of the copter. Therefore, there are three instances for each uncertainty type. In total, there are 24 uncertainty instances, affecting the flight.

ArduPlane (AP)[9] is an autonomous control system for fixed-wing aircraft, and it is instrumented with two **self-healing behaviors**. One is a rule-based policy for handling network disruption between an aircraft and its ground controller. When the response time from the controller is over a threshold, the aircraft is considered as disconnected with the controller, and then the behavior will control the aircraft to fly back and land on the launching place. The other self-healing behavior is a control algorithm used to avoid collision with a nearby aerial vehicle. AP uses four sensors and one actuator, the same with ArduCopter, to locate and manipulate an aircraft. Thus, its behaviors are also affected by 24 uncertainty instances.

ArduRover (AR)[10] is an open-source autopilot system for ground vehicles. It has two **self-healing behaviors**. One is a control algorithm for avoiding collisions. The other is a rule-based policy that helps a vehicle to drive back when it is disconnected with its radio controller. Totally, ArduRover employs three sensors (one accelerometer, one GPS, and one rangefinder) and one actuator (a motor) to control a vehicle. Since a ground vehicle runs on the ground, ArduRover only monitors and controls two dimensions of the vehicle, i.e., longitude and latitude. Thus, there are two instances for each type of uncertainty of the sensors and actuator. In total, ArduRover is affected by 14 uncertainty instances.

Adaptive Production Cell (APC) (Güdemann et al. 2006) is an autonomous manufacturing unit, which is comprised of three robots and three carts that deliver workpieces among the robots. A workpiece is to be processed in three steps, referred as three tasks for the robots. Every robot is equipped with three tools for accomplishing the three tasks. As it takes time for a robot to switch its tools, the three robots are configured to work together, and each of them only performs one of the tasks. APS is equipped with one **self-healing behavior**: a rule-based policy that reassigns tasks among robots to maintain the normal function of the production cell, in case one or multiple tools of a robot break. Due to inaccurate positions, delivered by the carts, and measured by the robots' sensors, APC is affected by 18 uncertainty instances: 3 carts × 3 instances of *Position Deviation* + 3 robots × 3 instances of *Position Accuracy*.

RailCab (RC) (Priesterjahn et al. 2013) is an autonomous railway system, whose function is to make rail vehicles drive in convoy to reduce energy consumptions. Driving in convoy

[9] http://ardupilot.org/plane/

[10] http://ardupilot.org/rover/

Table 3 Uncertainties in the Subject Systems

Sys.	Hardware	Uncertainty	Range	#Instances	Sys.	Hardware	Uncertainty	Range	#Instances
AC, AP, AR	Accelerometer	Acceleration Accuracy	(−9 mg, +9 mg)	3 for AC, AP 2 for AR	RC	Speed Sensor	Velocity Accuracy	(−0.1 m/s, 0.1 m/s)	2
		Nonlinearity	(−0.5%, +0.5%)			Eddy Current Sensor	Current Accuracy	(−1A, 1A)	8
	Motor	Rotation Deviation	(−0.3°, +0.3°)			Cab GPS	Position accuracy	(−2.5 m, +2.5 m)	2
		Acceleration Deviation	(−0.02 m/s^2, +0.02 m/s^2)				Velocity accuracy	(−0.05 m/s, +0.05 m/s)	2
	GPS	Position Accuracy	(−2.5 m, +2.5 m)			Cab Engine	Acceleration Deviation	(−0.1 m/s^2, 0.1 m/s^2)	2
		Velocity Accuracy	(−0.05 m/s, +0.05 m/s)				Rotation Deviation	(−1°, +1°)	2
AC, AP	Gyroscope	Angular Velocity Accuracy	(−0.3°/s, +0.3°/s)	3	MR	Laser Scanner	Direction Accuracy	(−5°, +5°)	2
	Barometer	Accuracy	(−150 Pa, +150 Pa)						
AR	Rangefinder	Accuracy	(−10 cm, +10 cm)	2					
APC	Cart	Position Deviation	(−0.1 m, +0.1 m)	9		Sonar Range Finder	Distance Accuracy	(−0.01 m, 0.01 m)	2
	Cell Monitor	Position Accuracy	(−0.05 m, 0.05 m)	9		Robot Motor	Rotation Deviation	(−3°, +3°)	2
							Acceleration Deviation	(−0.01 m/s^2, +0.01 m/s^2)	2

requires the vehicles to maintain a small distance between each other, and thus it is crucial to keep a correct speed and direction of all vehicles in convoy. RC employs two speed sensors and eight eddy current sensors to measure the speed and steering direction of a vehicle. The **self-healing behavior** of RC is a quantitative model-based algorithm, used to detects errors caused by malfunction of the speed sensors used by a vehicle. In case the errors are detected, the behavior will identify the fault and adapt the vehicle to use GPS rather than speed sensors to measure its speed. As shown in Table 3, the movement is affected by 18 uncertainty instances, arising from the 11 sensors and one actuator.

Mobile Robot (MB) (Steinbauer et al. 2005) (Brandstotter et al. 2007) is an autonomous robot control system for directing a robot to play soccer. In normal operation mode, MB controls three omnidirectional wheels to move the robot. Three **self-healing behaviors** are implemented in MB. Two of the behaviors are control algorithms that are used to detect the incorrect movement caused by the fault that a wheel becomes stuck or a wheel rotates freely, and make the robot still follow a desired trajectory in case the fault happens. The remaining self-healing behavior is a rule-based policy used to detect and restart malfunctioning software services. As there are strong dependencies among the services, the self-healing behavior has to find a correct order to stop and start involved services. In the system, a robot is equipped with a laser scanner to locate a soccer, a sonar range finder to measure the distance to the soccer, and a motor for movement. In total, eight uncertainty instances are impacting the behaviors of a robot, with two instances of *Direction Accuracy* from the scanner, two instances of *Distance Accuracy* from the range finder, and four uncertainty instances from the motor.

Testing these self-healing behaviors is challenging, as it is needed to decide when and which fault is to be introduced to test the self-healing behavior. As a fault may occur at any time during execution, the set of all possible test cases is huge. It could be infeasible to cover all cases. To effectively find cases in which a self-healing behavior will fail, we have proposed the executable model-based testing in our previous work (Ma et al. 2019a). In this approach, we test an SH-CPS against a test model, by executing them together, sending them the same testing stimuli, and comparing their consequent states. In this way, no test cases need to be generated before test execution. Additionally, by learning from the results of performed stimuli, reinforcement learning algorithms can be applied to learn the best policies of choosing stimulus to more effectively detect unexpected behaviors.

3.3.2 Test Models

By applying MoSH (a modeling framework for testing SH-CPS under uncertainty) (Ma et al. 2019a), the first author of this paper built the test models[11] for the selected six SH-CPSs. For each system, we first captured its main components (e.g., sensors, actuators, and controllers) as UML classes, and then specified the components' state variables and testing interfaces as properties and operations. For each component, we further specified its expected behaviors as state machines. Based on the requirement of each system (summarized in Table 4), we defined state invariants for all the states in the state machines.

Note that the state of an SH-CPS comprises the states of its components, and the behaviors of all components form the SH-CPS behavior. With a flattening algorithm (Ma et al. 2019b), the components' behaviors can be combined into a single state machine, representing the

[11] The test models are available at http://zen-tools.com/journal/TSHCPS_RL.html

Table 4 Requirements of the Subject Systems Used for Deriving State Invariants

Subject System	Requirement	Exemplary Invariant
ArduCopter (AC) ArduPlane (AP) ArduRover (AR)	To avoid crash and collision, the distance between a copter/plan/rover and another object (e.g., intruding aircraft or obstacle) should always be greater than zero.	$dis>0$ where *dis* represents the distance between a vehicle and an obstacle
Adaptive Production Cell (APC)	Keep the normal function of the production cell, and ensure that the produced workpiece is valid.	$Validity=1$ where *validity* is a state variable used to measure if a produced workpiece is valid
RailCab (RC)	To avoid collisions, the distance between two adjacent vehicles must be greater than the braking distance.	$dis>v_1^2-v_2^2/2a$ Where *dis* is the distance between two adjacent vehicles; v_1 and v_2 are their velocities; a is acceleration
Mobile Robot (MR)	Ensure a robot follow a desired trajectory, i.e., ensure the distance between a robot's desired position and actual position is within *d* centimeters. *d* is the size of the robot.	$\|P_{actual}-P_{target}\|_{dis}<d$ where P_{actual} and P_{target} are actual and expected positions of a robot

behavior of the SH-CPS. Table 5 presents descriptive statistics of the combined state machine for the six SH-CPSs.

As explained in Section 2, the specified test models are executable, and they are executed together with the SH-CPSs for testing. The last two columns of Table 5 show the average time taken to execute a test model alone and the time taken to execute an SH-CPS (software part) with simulators of sensors, actuators, and environment, for one episode. As the software of the system and the simulation models used by the simulators are complex, executing an SH-CPS is computationally expensive. Consequently, compared with executing a test model, it is much slower to execute an SH-CPS.

3.4 Tasks

To assess the failure detection abilities and scalabilities of the reinforcement learning algorithms, we apply them to test the six SH-CPSs, check the effectiveness and efficiency of the algorithms, and calculate their time and space cost. An algorithm's performance depends on its

Table 5 Descriptive Statistics of Behaviors of the Subject Systems

Category	Subject System	# States	# Transitions	# Uncertainties	Average Model Execution Time (second)	Average System Execution Time (min)
Real-world Systems	ArduCopter (AC)	432	1396	24	12	8
	ArduPlane (AP)	96	270	24	3	6
	ArduRover (AR)	140	650	12	8	10
Systems from literature	Adaptive Production Cell (APC)	1016	8512	18	15	3
	RailCab (RC)	2160	13,310	18	18	5
	Mobile Robot (MR)	1080	4656	8	15	3

capability of learning, while it also relies on the number of episodes (i.e., testing runs) that an algorithm can take to detect failures, as well as the range of each uncertainty.

On the one hand, the number of episodes determines the number of opportunities that an algorithm can take to try different operation invocations or uncertainty values. The more episodes an algorithm can take, the more samples of fragility the algorithm can obtain to learn the optimal policy, and thus the higher the probability it can detect a failure. Ideally, we should not limit the number of episodes an algorithm can take to find failures. However, testing an SH-CPS is computationally costly and time-consuming, as many simulators are involved in simulating its sensors, actuators, and environment. With eight CPU cores and 16 GB of memory, it takes a few minutes to perform one episode. Limited by current available computational resources, we evaluated the performance of the algorithms for 1000, 2000, 3000, 4000, and 5000 episodes.

On the other hand, the range of each uncertainty will affect its impact on an SH-CPS. For instance, if the range of an measurement error is extremely small, it will have little impact on the measurement. In contrast, if the measurement error is sufficiently large, it may lead to an incorrect measurement, which may increase the risk of abnormal behavior. The ranges presented in Table 3 were defined based on the product specifications of the sensors and actuators, whereas the actual ranges of measurement errors and actuation deviations could differ from the specifications. To account for the effect of the ranges, we chose to test each SH-CPS with 10 sets of ranges, which includes the set of ranges shown in Table 3 as the standard ranges, and nine sets of ranges derived by increasing or reducing the standard ranges by 10, 20, 30, 40, and 50%. We use 10 scales, i.e., 60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150%, to represents these 10 sets of ranges. The scale of 100% represents the standard range, 60% denotes the ranges reduced by 40%, and 150% means the ranges increased by 50%.

In summary, we applied 14 combinations of reinforcement learning algorithms to test six SH-CPSs with 10 uncertainty scales and five numbers of episodes ranging from 1000 to 5000. In total, there are 300 testing tasks (6 SH-CPSs × 10 uncertainty scales × 5 numbers of episodes). Due to the probabilistic policies used by the reinforcement learning algorithms, even for the same testing task, an algorithm may generate different results. To account for this randomness, each of the 300 testing tasks was performed 10 times by each combination of reinforcement learning algorithms.

3.5 Hypotheses and Variables

For Goal 1, we aim to evaluate the effectiveness and efficiency of reinforcement learning algorithms in the context of testing. For effectiveness, as the purpose of testing is to find failures in the system under test, we chose to use the Number of Detected Failures (NDF) as the metric. In addition, as it is preferable to cover more behaviors of the system under test, we selected State Coverage (SCov) and Transition Coverage (TCov) as two additional metrics for assessing the effectiveness. Regarding efficiency, it is related to effectiveness and cost. As time cost is the main concern for testing, we chose to use the average amount of time spent to detect a failure as the metric.

Based on the selected metrics, we formulate two kinds of null hypotheses:

1. Null hypothesis: given a maximum number of episodes (*ENUM*) and an uncertainty scale (*SCALE*), there is no significant difference in effectiveness (measured by State Coverage (SCov), Transition Coverage (TCov), and the Number of Detected Failures (*NDF*)) among the combinations of reinforcement learning algorithms.

H_0: $\forall APP_i, APP_j \in APP\_SET, \forall\, SCALE_s \in SCALE\_SET, \forall\, ENUM_e \in ENUM\_SET$:

$$Effect(APP_i, SCALE_s, ENUM_e) = Effect\left(APP_j, SCALE_s, ENUM_e\right)$$

Alternative hypothesis,

H_1: $\exists APP_i, APP_j \in APP\_SET, \exists\, SCALE_s \in SCALE\_SET, \exists\, ENUM_e \in ENUM\_SET$:

$$Effect(APP_i, SCALE_s, ENUM_e) \neq Effect\left(APP_j, SCALE_s, ENUM_e\right)$$

APP_i: one of the 14 testing approaches

APP_SET: the set of the 14 testing approaches, $\{APP_{Q\_A3C}, APP_{Q\_TRPO}, APP_{Q\_UPO}, APP_{Q\_PPO}, APP_{Q\_DDPG}, APP_{S\_ACKTR}, APP_{S\_DDPG}, APP_{S\_A3C}, APP_{S\_ACER}, APP_{S\_PPO}, APP_{S\_TRPO}, APP_{S\_UPO}, APP_{Q\_ACKTR}, APP_{Q\_ACER}\}$[12]

$SCALE\_SET$: the set of uncertainty scales, {60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150%}

$ENUM\_SET$: the set of numbers of episodes, {1000, 2000, 3000, 4000, 5000}

Effect represents an effectiveness metric, which could be *SCov*, *TCov*, or *NDF*

SCov is the percentage of states that are covered by a number of episodes. It is calculated by: $SCov(APP_i, SCALE_s, ENUM_e) = \frac{\left|\cup_{k=1}^{ENUM_e} S_{k,APP_i,SCALE_s}\right|}{|S_{all}|}$, where $S_{k,APP_i,SCALE_s}$ represents the set of states, visited by APP_i in the kth episode, under uncertainty scale $SCALE_s$; S_{all} represents the set of all states in a test model.

TCov is the percentage of covered transitions. Similar with *SCov*, *TCov* is calculated by $TCov(APP_i, SCALE_s, ENUM_e) = \frac{\left|\cup_{k=1}^{ENUM_e} T_{k,APP_i,SCALE_s}\right|}{|T_{all}|}$, where $T_{k,APP_i,SCALE_s}$ is the set of transitions, visited by APP_i in the kth episode, under uncertainty scale $SCALE_s$; T_{all} represents the set of all transitions.

NDF is calculated by:$NDF(APP_i, SCALE_s, ENUM_e) = \sum_{k=1}^{ENUM_e} FD_{k,APP_i,SCALE_s}$, where $FD_{i,APP_i,SCALE_s}$ denotes whether a failure is detected by APP_i in the kth episode, under uncertainty scale $SCALE_s$. $FD_{i,APP_i,SCALE_s}$ equals 1 if a failure is detected, otherwise, 0.

2. Null hypothesis: given an *ENUM* and a *SCALE*, there is no significant difference in efficiency (measured by an efficiency measure *EFF*), among the combinations of reinforcement learning algorithms.

H_0: $\forall APP_i, APP_j \in APP\_SET, \forall\, SCALE_s \in SCALE\_SET, \forall\, ENUM_e \in ENUM\_SET$:

$$EFF(APP_i, SCALE_s, ENUM_e) = EFF\left(APP_j, SCALE_s, ENUM_e\right)$$

Alternative hypothesis,

H_1: $\exists APP_i, APP_j \in APP\_SET, \exists\, SCALE_s \in SCALE\_SET, \exists\, ENUM_e \in ENUM\_SET$:

$$EFF(APP_i, SCALE_s, ENUM_e) \neq EFF\left(APP_j, SCALE_s, ENUM_e\right)$$

[12] Q represents Q-learning and S represents SARSA.

EFF is calculated by: $EFF(APP_i, SCALE_s, ENUM_e) = \frac{\sum_{k=1}^{ENUMe} TCost_{k,APP_i,SCALE_s}}{NDF(APP_i,SCALE_s,ENUM_e)}$, where $\sum_{k=1}^{ENUM_e} TCost_{k,APP_i,SCALE_s}$ is the total amount of time taken by APP_i for the number of episodes of $ENUM_e$.

For Goal 2, we evaluated the time and space costs of each combination of reinforcement learning algorithms. For each testing task, we measured the following two variables:

- Time Cost (TCost): $TCost(APP_i, SCALE_s, ENUM_e) = \frac{\sum_{k=1}^{ENUM_e} TC_{ostk,APP_i,SCALE_S}}{ENUM_e}$
- Space Cost (SCost): $SCost(APP_i, SCALE_s, ENUM_e) = \max_{1 \le k \le ENUM_e} SCost_{k,APP_i,SCALE_s}$, where $SCost_{k,APP_i,SCALE_s}$ denotes the memory usage of APP_i for the kth episode, under uncertainty scale $SCALE_s$.

In summary, the empirical study involves three independent variables and six dependent variables. Table 6 summarizes their values and mapping to the goals.

3.6 Statistical Tests

Table 7 summarizes the statistical tests and related variables used to evaluate the effectiveness, efficiency, time cost, and space cost of the 14 combinations of reinforcement learning algorithms. We first tested the normality of the samples of dependent variables (*SCov*, *TCov*, *NDF,* and *EFF*), using the Shapiro-Wilk test (Royston 1995) with a significance level of 0.05. Test results showed that the samples are not normally distributed. Therefore, we chose to use non-parametric statistics, the Kruskal-Wallis test (Kruskal and Wallis 1952) and the Dunn's test (Dunn 1964) in conjunction with the Benjamini-Hochberg correction (Benjamini and Hochberg 1995), to check statistical significances, and applied the Vargha and Delaney statistics (Vargha and Delaney 2000) to measure effect sizes.

For the samples of dependent variables, we first applied the Kruskal-Wallis test to check whether there are significant differences in these variables among the 14 combinations of algorithms. If the Kruskal-Wallis test indicates there are significant differences (i.e., a p value is less than 0.05), we further performed the Dunn's test in conjunction with the Benjamini-Hochberg correction to evaluate the significance of the difference of each pair of data groups.

For each data groups pair, we also applied the Vargha and Delaney statistics $\widehat{A}_{12}$ to measure the effect size, which reveals the probability that an approach A has higher *SCov*, *TCov*, *NDF,* or *EFF* than the other approach B. If $\widehat{A}_{12}$ equals to 0.5, then the two approaches perform equally well. If $\widehat{A}_{12}$ is greater than 0.5, then A has a higher chance to perform better, and vice versa.

4 Experiment Execution

We introduce the hyperparameter settings of the reinforcement learning algorithms in Section 4.1 and the experiments' execution process in Section 4.2.

4.1 Hyperparameter Tuning

Although reinforcement learning algorithms have demonstrated great learning abilities in multiple fields (Mnih et al. 2015; Lillicrap et al. 2015; Kober et al. 2013), the success of a particular learning algorithm depends upon the joint tuning of the model structure and optimization procedure (Jaderberg et al. 2017). Both of them are controlled by several hyperparameters, such as a neural network's structure, learning rate, loss function, and the number of episodes. The hyperparameters impact the whole learning process, and must be tuned to fully unlock an algorithm's potential. However, hyperparameter tuning is computationally expensive and time-consuming. In the context of software testing, testers may not have a sufficient time budget to tune an algorithm for every system under test. It will be helpful to have a hyperparameter setting that can achieve relatively good performance for a wide range of systems.

Reinforcement learning researchers have recommended several default hyperparameter settings (Henderson et al. 2018). However, these settings were tuned for playing computer games, which are different from the testing problem. Due to the high computational cost of hyperparameter tuning and limited computational resources we have, we could not afford to use all systems with all variables' settings to tune the hyperparameter. Among the six selected SH-CPSs, AP is the simplest, with the least number of states and transitions, while RC is the most complex one, and AC is a moderate one. We chose to use these three systems with diverse complexities to do the tuning, to make the selected systems more representative. For the value of the uncertainty scale and the number of episodes, we chose a moderate setting for tuning, i.e., the uncertainty scale of 100% and the number of 3000, so as to avoid achieving a hyperparameter setting performing well only in extreme cases.

We applied the Population-Based Training (PBT) method (Jaderberg et al. 2017) for tuning. Compared with sequential optimization or parallel grid/random search, PBT can focus on the hyperparameter space that has a higher chance of producing good results, and thus more efficiently find a better hyperparameter setting. For each reinforcement learning algorithm used in the experiment (Section 3.2), PBT was allowed to take maximally 1000 iterations to find the optimal hyperparameter setting. During the search, the three systems (AC, AP, and RC) were used to evaluate the performance of a setting, with the uncertainty scale of 100% and the number of episodes of 3000. The setting that leads to the highest fragility was regarded as the optimal solution. Table 8 presents the optimal hyperparameter setting we found for each algorithm.

4.2 Execution Process

As explained in Section 3, we applied 14 combinations of reinforcement learning algorithms to test six SH-CPSs with 10 uncertainty scales and five settings of episodes numbers. The experiment was conducted on Abel, a cluster at the University of Oslo.[13] Each testing job was run on eight nodes with 32 GB RAM. The whole empirical study took more than six months' worth of execution time.

We measured the state coverage (*SCov*), transition coverage (*TCov*), number of detected failures (*NDF*), time cost (*TCost*), and space cost (*SCost*) for each approach and each testing task. The approaches' efficiencies (*EFF*s) were further calculated, using *NDF* and *TCost*. At

[13] http://www.uio.no/english/services/it/research/hpc/abel/

Table 6 Independent and Dependent Variables

Variable Type	Variable Name	Value	Mapping to Goals
Independent Variable	*APP*	APP_{Q_A3C}, APP_{Q_TRPO}, APP_{Q_UPO}, APP_{Q_PPO}, APP_{Q_DDPG}, APP_{S_ACKTR}, APP_{S_DDPG}, APP_{S_A3C}, APP_{S_ACER}, APP_{S_PPO}, APP_{S_TRPO}, APP_{S_UPO}, APP_{Q_ACKTR}, APP_{Q_ACER}	Goal 1, Goal 2
	SCALE	60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150%	
	ENUM	1000, 2000, 3000, 4000, 5000	
Dependent Variable	*SCov*	Real Number	Goal 1
	TCov	Real Number	
	NDF	Integer Number	
	EFF	Real Number	
	TCost	Real Number	Goal 2
	SCost	Real Number	

last, we applied statistical tests to assess the differences of the measurements among the 14 approaches.

5 Experiment Results

This section shows the results of the empirical study. Sections 5.1 and 5.2 present the effectiveness and efficiency of the 14 combinations of reinforcement learning algorithms, and Section 5.3 analyses their time and space costs.

5.1 Effectiveness

For most of the testing tasks, the 14 testing approaches managed to cover all the states and transitions of the SH-CPS under test, i.e., the state coverage (*SCov*) and transition coverage (*TCov*) equal to 100%. The exceptions are the tasks for testing RC, APC, and MR. For testing APC and MR with 1000 episodes, only 74.2% and 69.4% transitions were covered, on average. When the number of episodes increased to above 2000, all of the approaches managed to cover all transitions. For testing RC, none of the 14 approaches achieved 100% transition coverage, and only a few approaches got 100% state coverage in very few cases, as RC has huge sets of states and transitions (Table 5). As an example, Fig. 3 presents the box plots of the state and transition coverages of the approaches for testing RC. The *p*-values of the

Table 7 Overview of Statistical Tests with Goals

Goal	Description	Dependent Variable	Statistical Test
1	G1.1. For each pair of approaches, compare their *SCov*, *TCov*, and *NDF*s	*SCov*, *TCov*, *NDF*	The Kruskal-Wallis test The Dunn's test
	G1.2. For each pair of approaches, compare their *EFF*s	*EFF*	The Vargha and Delaney statistics
2	T2.1. Evaluate *TCost* for each approach	*TCost*	N/A
	T2.2. Evaluate *SCost* for each approach	*SCost*	

Kruskal-Wallis test in terms of the state and transition coverages for all the testing tasks are greater than 0.1, thereby indicating no significant difference among the 14 approaches regarding the coverages.

Next, we assess the actual failure detection ability of the approaches. There are 41 failures detected in the six SH-CPSs, with seven failures detected in AC, eight failures detected in AR, eight failures detected in AP, five failures detected in APC, eight failures detected in RC, and five failures detected in MR. These 41 failures correspond to 41 states in which their invariants were violated when the six systems were being tested with simulated sensors and actuators. Examples of these failures include a collision between a copter and an intruding vehicle, a crash of a plane on the ground, and a collision between a rover and an obstacle. Table 9 presents the average number of detected failures (*NDF*) for the 14 testing approaches, under 10 uncertainty scales and five numbers of episodes.

As shown in the table, APP_{Q_UPO} managed to detect the most failures. On average, it detected 3.4 failures in a testing task. APP_{S_UPO} performed slightly worse, with 3.3 failures detected averagely. In contrast, the other 12 approaches only detected 1.7 failures, on average, and only detected three failures or less in most of the testing tasks.

We first conducted the Kruskal-Wallis test to determine whether there are significant differences in the *NDF*s among the 14 approaches. The *p* value of the Kruskal-Wallis test is less than 10^{-13}, and thus significant differences do exist. Afterward, we applied the Dunn's test together with the Benjamini-Hochberg correction to check the significance of the difference in *NDF*s between each pair of approaches. The effect size of the difference was also evaluated, using the Vargha and Delaney statistics $\widehat{A}_{12}$. Since APP_{Q_UPO} detected the most failures, we focused on checking if this superiority is statistically significant.

For each of the other 13 testing approaches, denoted as APP_c, we checked the *p* value of the Dunn's test, corresponding to the pair of APP_{Q_UPO} and APP_c. If the *p* value is over 0.05, the two testing approaches are considered to be performing equally well. Otherwise, we further examined the Vargha and Delaney statistics $\widehat{A}_{12}$, using the *NDF*s of the two approaches. If $\widehat{A}_{12}$ is above 0.5 for the pair of APP_{Q_UPO} and APP_c, it means APP_{Q_UPO} has a higher chance to detect more failures, and thus APP_{Q_UPO} is considered to be superior to APP_c. Otherwise, APP_{Q_UPO} is considered to be worse.

In over 239 (out of 300) testing jobs, APP_{Q_UPO} significantly outperformed the other 12 testing approaches, except APP_{S_UPO}. APP_{Q_UPO} and APP_{S_UPO} performed equally in 279 jobs; APP_{S_UPO} beat APP_{Q_UPO} in 2 jobs and APP_{Q_UPO} was superior in the other 19 jobs. Table 10 in Appendix 1 presents detailed results. When the maximum number of episodes (*ENUM*) is 1000, all testing approaches performed almost the same, while, as *ENUM* increases, APP_{Q_UPO} and APP_{S_UPO} exceeded the others. When the uncertainty scale (*SCALE*) is above 120%, APP_{Q_UPO} detected more failures than APP_{S_UPO} did in nine jobs, and they performed almost on the same level in other cases.

5.2 Efficiency

We evaluate the efficiency of the reinforcement learning algorithms, to find the combination of algorithms that takes the shortest time to detect failures. Figure 4 shows the time taken by the algorithms to execute an SH-CPS from its initial state to a final state once. Particularly, the time cost includes the time taken to select operations and uncertainty values, generate test input, invoke corresponding testing interfaces, execute the system, evaluate the fragility of the

Table 8 Overview of Hyperparameter Settings

Algorithm	Hyperparameter	Value	Algorithm	Hyperparameter	Value
Q-learning	Learning Rate	0.1	SARSA	Learning Rate	0.1
	Discount Rate	0.98		Discount Rate	0.98
	ε-greedy	0.2		ε-greedy	0.2
A3C	Discount Rate	0.99	ACER	Discount Rate	0.99
	#Hidden Layers	3		#Hidden Layers	2
	#Hidden Neurons	32		#Hidden Neurons	96
	Activation Function	ReLU		Activation Function	ReLU
	Optimizer	RMSProp		Optimizer	RMSProp
	Learning Rate	0.1		Learning Rate	0.001
	Batch Size	1000		Batch Size	500
	#Epochs	1		#Epochs	1
PPO	Discount Rate	0.9	TRPO	Discount Rate	0.9
	#Hidden Layers	3		#Hidden Layers	3
	#Hidden Neurons	32		#Hidden Neurons	96
	Activation Function	Tanh		Activation Function	Tanh
	Optimizer	Adam		Optimizer	Adam
	Learning Rate	0.001		Learning Rate	0.001
	Clip	0.3		Max KL	0.05
	Batch Size	500		Batch Size	2000
	#Epochs	10		#Epochs	50
ACKTR	Discount Rate	0.9	DDPG	Discount Rate	0.9
	#Hidden Layers	2		#Hidden Layers	3
	#Hidden Neurons	32		#Hidden Neurons	32
	Activation Function	Tanh		Activation Function	ReLU
	Optimizer	Kfac		Optimizer	Adam
	Learning Rate	0.01		Learning Rate	0.0001
	Max KL	0.01		Batch Size	1000
	Batch Size	2500		#Epochs	50
	#Epochs	1			
UPO	Discount Rate	0.99			
	Activation Function	Tanh			
	#Hidden Layers	3			
	#Hidden Neurons	96			

consequent states, and use the fragility to update the Q function and uncertainty policy. On average, the testing approaches took less than 150 s to complete one episode. The differences among the average execution times of the different testing approaches are small, within 10 s. However, for different SH-CPSs, *SCALE*s, and *ENUM*s, the execution time varies a lot, ranging from 53 s to 480 s. Compared with the approaches using SARSA, the approaches with Q-learning took less time to perform one episode.

As the time taken to execute the system depends on the system's implementation, rather than the performance of the algorithms, Fig. 5 presents the time cost, excluding the time spent for executing the system. On average, the algorithms took about 12 s in one episode. Consistent with the trend revealed by Figs. 4 and 5 also shows that SARSA related approaches took a longer time to perform an episode. In general, APP_{Q_A3C} incurred the least time cost, though the differences are within 5 s.

Based on the time costs and the number of detected failures (*NDF*), shown in Section 5.1, we calculated the efficiency measure (*EFF*), shown in Fig. 6. Unsurprisingly, APP_{Q_UPO} took the least amount of time to detect a failure, since all testing approaches took a similar amount of time and APP_{Q_UPO} detected the most failures within this time. Averagely, APP_{Q_UPO} took

64.5 h to detect a failure, which is less than half of the average time taken by APP_{Q_A3C} (the least efficient approach) for failure detection. On average, APP_{Q_UPO} took 52% less time than the other approaches to detect a failure.

To evaluate the significance of the differences, we conducted the Kruskal-Wallis test. The p value of the test is less than 10^{-10}, and thus there are significant differences in the EFFs among different testing approaches. We further applied the Dunn's test together with the Benjamini-Hochberg correction to examine if EFFs of APP_{Q_UPO} are significantly smaller than $EFFs$ of the other approaches. The Vargha and Delaney statistics $\widehat{A}_{12}$ was used to assess the effect size. Compared with APP_{S_UPO}, APP_{Q_UPO} took significantly less time for failure detection in 98 jobs, more time in one job, and performed equally well in 201 jobs. For the other testing approaches, APP_{Q_UPO} was significantly more efficient in over 238 jobs. Table 11 in Appendix 2 presents more details.

5.3 Scalability

We first assess the tendencies of time and space costs of the 14 testing approaches for learning the policy of choosing operation invocations, i.e., learning how to trigger the outgoing transitions of each state defined in a test model to maximize the fragility of the SH-CPS under test. Figure 7 shows the average time cost of each testing approach per episode, and Fig. 8

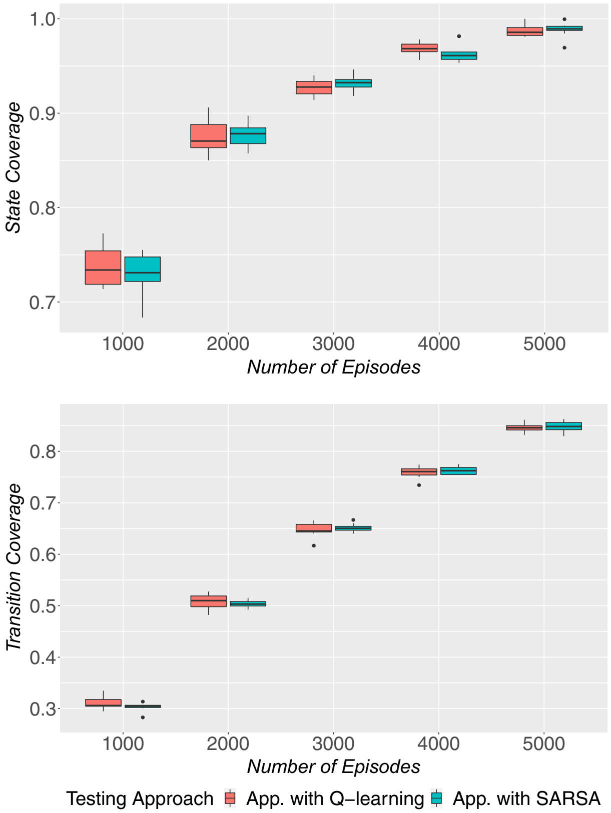

Fig. 3 State and Transition Coverages for RC

Table 9 Average Numbers of Detected Failures (*NDF*) by the 14 testing Approaches

SCALE	ENUM	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14
60%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.3**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**2.3**	1.0	1.0
	3000	1.7	1.8	**3.6**	1.6	1.7	1.7	1.7	1.7	1.7	1.5	1.7	3.5	1.7	1.7
	4000	1.9	2.0	**4.8**	2.0	1.9	2.0	2.0	2.1	2.1	2.1	2.0	4.7	1.9	2.0
	5000	2.8	2.8	**6.0**	2.8	2.8	2.8	2.8	2.9	2.8	2.9	2.9	5.9	2.8	2.8
70%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.3**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**2.3**	1.0	1.0
	3000	1.7	1.6	3.4	1.6	1.7	1.7	1.7	1.7	1.6	1.5	1.6	**3.5**	1.7	1.8
	4000	1.9	2.0	**4.9**	2.0	2.0	2.0	2.0	2.0	2.1	2.1	2.1	4.6	2.0	2.0
	5000	2.8	2.8	**5.9**	2.8	2.8	2.8	2.8	2.9	2.8	2.9	2.8	5.8	2.8	2.8
80%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.4**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.3	1.0	1.0
	3000	1.8	1.8	**3.6**	1.8	1.7	1.5	1.7	1.7	1.7	1.7	1.6	3.4	1.7	1.6
	4000	2.0	2.0	**4.8**	2.0	2.0	2.1	2.0	2.1	2.0	2.0	2.0	4.6	2.0	2.0
	5000	2.8	2.8	5.9	2.8	2.8	2.8	2.8	2.9	2.8	2.9	2.8	**6.0**	2.8	2.8
90%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.3**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.1	1.0	1.0
	3000	1.7	1.7	**3.6**	1.7	1.7	1.7	1.7	1.6	1.6	1.6	1.6	3.3	1.7	1.6
	4000	2.0	2.0	**4.7**	2.1	2.0	2.0	1.9	2.0	2.0	2.0	1.9	**4.7**	2.0	2.0
	5000	2.8	2.8	**6.0**	2.8	2.8	2.8	2.9	2.9	2.8	2.9	2.8	5.9	2.8	2.8
100%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.9	**1.0**	**1.0**
	2000	1.0	1.0	**2.3**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.2	1.0	1.0
	3000	1.6	1.6	**3.4**	1.7	1.8	1.7	1.6	1.6	1.5	1.7	1.8	3.3	1.7	1.7
	4000	1.9	2.0	**4.7**	2.0	1.9	2.1	2.1	2.0	2.0	2.1	2.0	4.4	1.9	1.9
	5000	2.8	2.8	**5.6**	2.8	2.8	2.8	2.9	2.8	2.8	2.8	2.8	5.5	2.8	2.8
110%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	2.2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	**2.3**	1.0	1.0
	3000	1.8	1.7	**3.5**	1.7	1.7	1.5	1.6	1.5	1.6	1.7	1.6	3.3	1.7	1.6
	4000	1.9	2.0	**4.6**	2.0	2.0	2.0	2.1	2.0	2.0	2.0	2.0	4.4	2.0	1.9
	5000	2.8	2.8	**5.6**	2.8	2.8	2.8	2.9	2.8	2.8	2.8	2.8	5.5	2.8	2.7
120%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.4**	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.1	1.0	1.0
	3000	1.6	1.7	**3.4**	1.6	1.7	1.7	1.6	1.6	1.7	1.5	1.6	3.2	1.6	1.6
	4000	2.0	2.1	**4.6**	2.1	2.0	1.9	2.0	1.9	2.0	2.0	2.1	4.3	2.0	1.9
	5000	2.7	2.8	**5.6**	2.8	2.8	2.8	2.9	2.7	2.7	2.8	2.8	5.4	2.8	2.6
130%	1000	0.9	1.0	1.0	1.0	1.0	1.0	1.0	**1.1**	**1.1**	1.0	1.0	0.9	1.0	1.0
	2000	1.0	1.0	**2.3**	1.0	1.0	1.0	1.0	1.1	1.1	1.0	1.0	2.0	1.0	1.0
	3000	1.6	1.6	**3.5**	1.7	1.6	1.5	1.7	1.7	1.7	1.7	1.6	3.1	1.6	1.7
	4000	1.9	2.0	**4.5**	2.0	2.0	1.8	2.0	2.0	2.0	2.0	2.1	4.2	2.0	1.9
	5000	2.6	2.8	**5.7**	2.8	2.8	2.7	2.9	2.7	2.7	2.8	2.9	5.2	2.7	2.6
140%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.9	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.2**	1.0	1.0	1.0	1.0	1.1	1.1	1.0	1.0	1.9	1.0	1.0
	3000	1.5	1.7	**3.5**	1.7	1.7	1.6	1.7	1.6	1.6	1.6	1.6	3.1	1.7	1.5
	4000	1.9	2.0	**4.6**	1.9	2.0	1.9	2.1	1.9	2.0	2.0	2.0	4.2	1.8	1.9
	5000	2.6	2.8	**5.7**	2.7	2.8	2.6	2.9	2.7	2.6	2.7	2.9	5.1	2.7	2.6
150%	1000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	0.9	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
	2000	1.0	1.0	**2.2**	1.0	1.0	1.0	1.0	1.1	1.1	1.0	1.0	2.0	1.0	1.0
	3000	1.6	1.6	**3.4**	1.6	1.7	1.6	1.6	1.6	1.7	1.6	1.7	3.1	1.7	1.6
	4000	1.9	2.0	**4.7**	1.9	2.0	1.9	2.0	1.9	1.9	2.0	2.0	4.1	1.9	1.9
	5000	2.6	2.8	**5.6**	2.7	2.8	2.6	2.8	2.6	2.6	2.7	2.9	5.2	2.6	2.6
Average		1.7	1.7	**3.4**	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	3.3	1.7	1.7

*#1: $\mathrm{APP}_{\mathrm{Q\_A3C}}$, #2: $\mathrm{APP}_{\mathrm{Q\_TRPO}}$, #3: $\mathrm{APP}_{\mathrm{Q\_UPO}}$, #4: $\mathrm{APP}_{\mathrm{Q\_PPO}}$, #5: $\mathrm{APP}_{\mathrm{Q\_DDPG}}$, #6: $\mathrm{APP}_{\mathrm{S\_ACKTR}}$, #7: $\mathrm{APP}_{\mathrm{S\_DDPG}}$, #8: $\mathrm{APP}_{\mathrm{S\_A3C}}$, #9: $\mathrm{APP}_{\mathrm{S\_ACER}}$, #10: $\mathrm{APP}_{\mathrm{S\_PPO}}$, #11: $\mathrm{APP}_{\mathrm{S\_TRPO}}$, #12: $\mathrm{APP}_{\mathrm{S\_UPO}}$, #13: $\mathrm{APP}_{\mathrm{Q\_ACKTR}}$, #14: $\mathrm{APP}_{\mathrm{Q\_ACER}}$

presents their average space costs. In the figures, the systems are shown in the increasing order of numbers of states and transitions, i.e., AP has the least states and transitions, and RC has the most. As shown in the figures, all costs exhibit the same tendency: the more states and transitions an SH-CPS has, the more time and space a testing approach took to learn the optimal policy of invoking operations for failure detection. For the simplest subject system, AP, which contains 96 states and 270 transitions, the 14 testing approaches took about 10 s to perform one episode, and used 5 GB memory space on average. In contrast, for the most complex system, RC, with 2160 states and 13,310 transitions, the testing approaches' time and space costs raised to about 70 s and 15 GB respectively.

For the second task, the algorithms need to learn the policy of choosing uncertainty values that can impede the SH-CPSs and lead to failures. Figures 9 and 10 present tendencies of their time and space costs as the number of involved uncertainties increase, where the systems are shown in the increasing order of the number of uncertainties. As one can see from the figures, the time and space costs remained at the same level for the six SH-CPSs with varying numbers of uncertainties. Because the policy based algorithms employ Artificial Neural Network (ANN) to select actions, they do not need to store the Q values of all actions for each state. Consequently, their time and space costs are fixed as long as the architectures of the ANNs are not changed. In Appendix 3, Table 12 presents the detailed time and space cost of each testing approach for each SH-CPS.

6 Discussion

This section discusses the experiment results about effectiveness in Section 6.1, efficiency in Section 6.2, scalability in Section 6.3 and alternative approaches in Section 6.4.

6.1 Effectiveness

Based on the results of the effectiveness, we can conclude that the combination of Q-learning and UPO is the most effective approach that detected the most failures in the six SH-CPSs, even though the 14 testing approaches achieved similar state and transition coverages. As shown in Fig. 11, when *ENUM* equals 1000, the 14 combinations of the algorithms performed at the same level. When the algorithms only had 1000 episodes to find failures, they could not collect sufficient experience from the executions, and just detected one failure on average. As the *ENUM* increased, the algorithms had more chances to explore the states of the system under test, with diverse operation invocations and uncertainty values. Consequently, the algorithms had more data to optimize their policies, and applied them to detect more failures. However, the increasing tendencies of *NDF* are different for the 14 testing approaches. The approaches with UPO tend to detect more failures than the approaches using A3C, ACER, ACKTR, DDPG, PPO, and TROP. Because the algorithms, like A3C and ACER, have to learn a value function to evaluate their policies and then update the policies based on the value function, many episodes were needed to obtain data for learning the value function, whenever the policy is updated. In contrast, UPO explores the space of policy directly using a probabilistic policy, which keeps UPO on trying different sequences of uncertainty values. Whenever it finds an uncertainty sequence that leads to a higher return, it updates its policy to increase the probability of selecting such a sequence. Therefore, UPO eliminates the cost of learning value functions, and potentially covers more promising sequences of uncertainty values for failure detection.

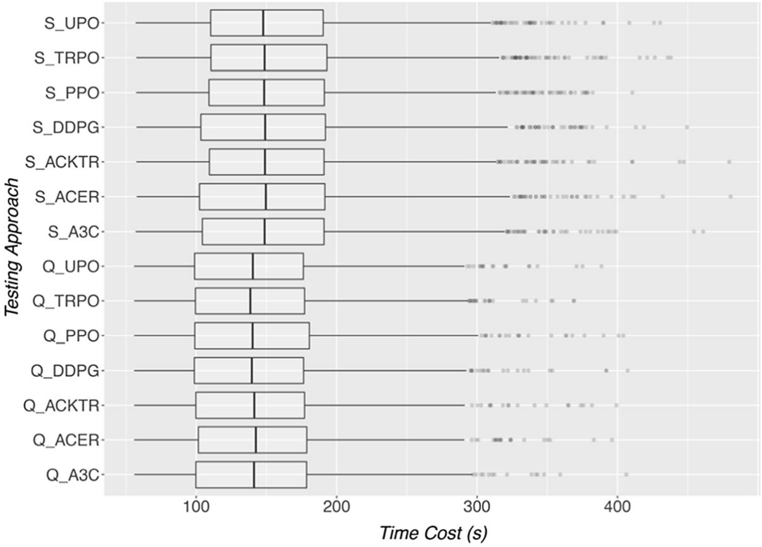

Fig. 4 Total Time Cost for One Episode

Compared with *ENUM*, *SCALE* has less effect on the *NDF*. As shown in Fig. 12, the average *NDF*s of the different testing approaches decrease slightly, as *SCALE* increases. A higher *SCALE* leads to larger uncertainty values, which may cause a greater impact on system behaviors and make them more likely to fail. However, a large *SCALE* also broadens the space of uncertainty values, which makes it more difficult to find the optimal sequence of uncertainty values. Due to these two reasons, *SCALE* is only slightly affected by the *NDF*. It should also note that the ranges of uncertainty values have great impact on the performance of the testing approaches. Sufficient knowledge is required to specify the ranges correctly prior to applying the testing approaches.

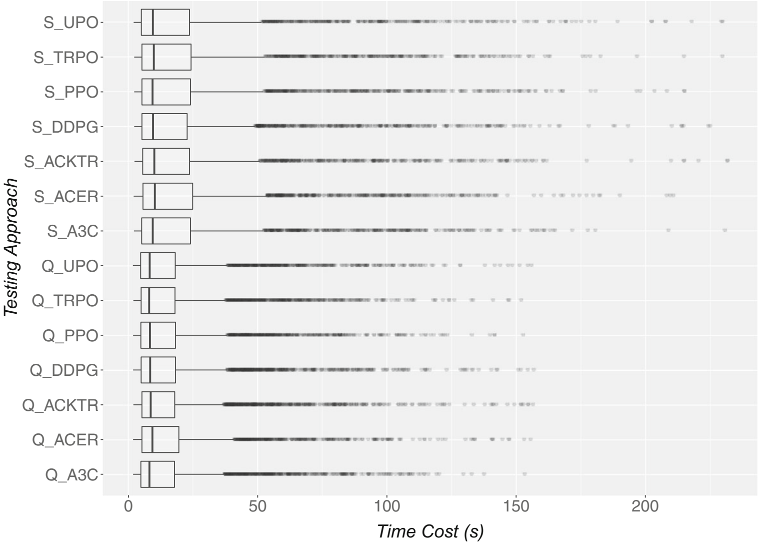

Fig. 5 Algorithm Related Time Cost for One Episode

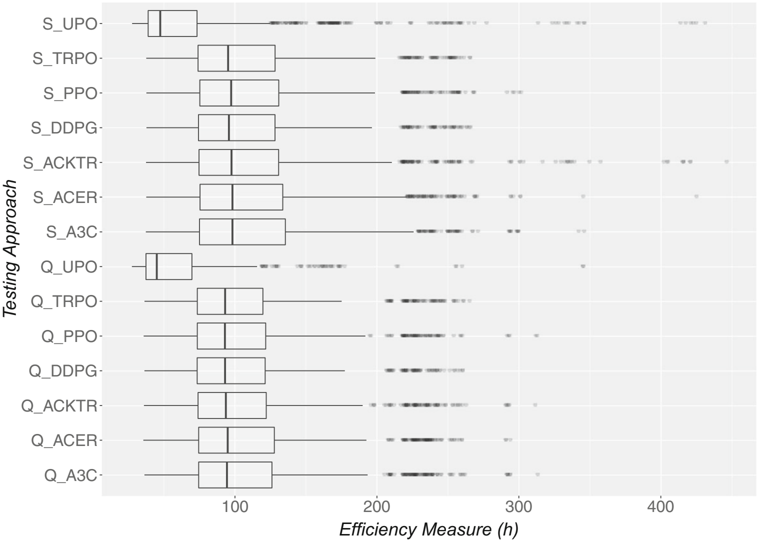

Fig. 6 Testing Approaches' Efficiencies for 300 Testing Jobs

6.2 Efficiency

Based on the results of the efficiency (Section 5.2), we know that the combination of Q-learning and UPO took the least amount of time to detect a failure. As shown in Fig. 5, on average, the algorithms took about 12 s to perform one episode. The difference in the average time costs among different algorithms is within 5 s. For operation invocations, both SARSA and Q-learning aim to learn the optimal Q function, based on which the algorithms find the optimal policy. While the Q function is being updated, SARSA has to follow its current policy (Eq. (4)), whereas Q-learning only needs to find the maximum Q value of consequent states (Eq. (5)). Therefore, the computational complexity of Q-learning is less than SARSA's. For introducing uncertainty, all of the policy optimization algorithms calculate a standard gradient or natural gradient, and apply gradient descent methods to optimize their policies. Consequently, their computational complexities are similar and mainly determined by the architecture of ANNs, used as the policy and value function in these algorithms. Since, in this experiment, the ANNs used by the algorithms have almost the same number of layers and the same number of neurons in each layer (Table 8), the algorithms had very similar time costs.

6.3 Scalability

The results in Section 5.3 reveal that the time and space costs of learning the policy of selecting uncertainty values remain in the same order of magnitude for testing SH-CPSs with diverse complexities. In contrast, the costs of learning the policy of invoking operations are rising as the numbers of states and transitions of the system under test increase. Since the policy optimization methods were used for uncertainties and they take advantage of ANNs to approximate their policies and value functions, the computational costs of these algorithms are determined by the architecture of the ANNs and the optimizer to improve the ANNs. Alternatively, value function methods, i.e., Q-learning and SARSA used for operation invocations, have to store the Q value for

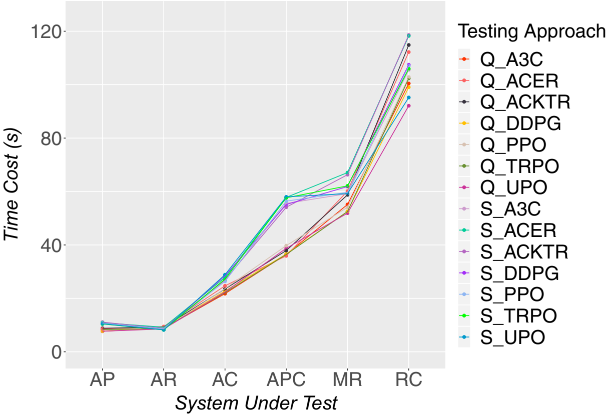

Fig. 7 Average Time Cost for Choosing and Invoking Operations

each pair of state and transition explicitly. As the number of states and transitions increases, such methods will take more space and time to store and process the Q values. This could be a potential scalability issue that limits the maximum numbers of states and transitions of the SH-CPS under test. One approach to resolve this issue is to use ANNs to approximate the Q value. However, as explained in Section 3.2, for testing an SH-CPS, the candidate operation invocations are not fixed. They will change when the system switch from one state to another. As the inputs of ANNs have to be fixed, we have to train an ANN for each operation to predict its Q value. Further study is needed to determine whether multiple ANNs could be trained together efficiently to find the optimal policy for operation invocations.

6.4 Alternative Approaches

The objective of testing an SH-CPS under uncertainty is to find a sequence of operation invocations and a sequence of uncertainty values that can work together to make an SH-CPS fails to behave as expected. We formulate this testing problem as an optimization problem, that is to find the optimal policies of choosing operation invocations and uncertainty values to maximize the chance of detecting failures. Through a trial and error process, reinforcement

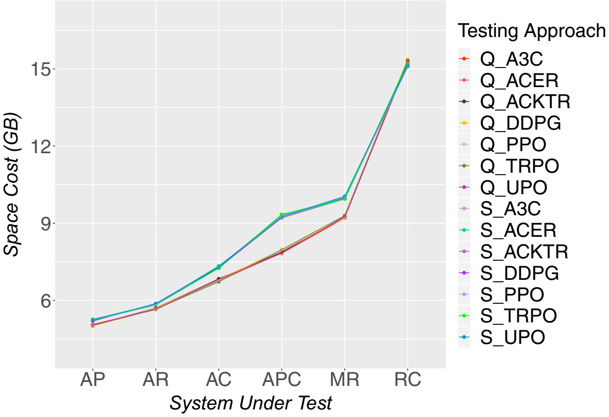

Fig. 8 Average Space Cost for Choosing and Invoking Operations

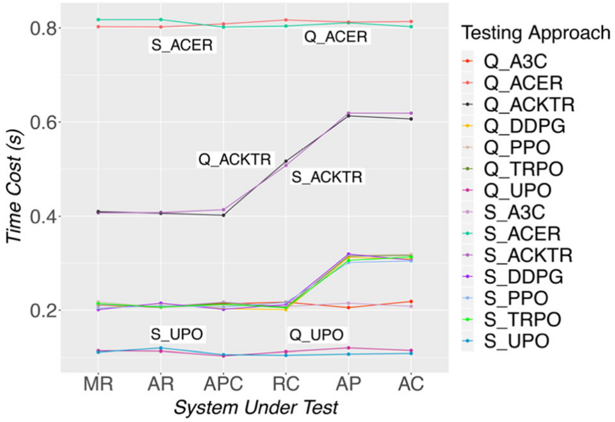

Fig. 9 Average Time Cost for Selecting and Introducing Uncertainties

learning can learn how to select testing stimuli to reach the highest fragility and reveal a failure. Results from our previous work also demonstrated the effectiveness of RL in solving this testing problem (Ma et al. 2019b).

As explained in Section 2.3, since the number of possible combinations of uncertainty values is huge, we chose to decompose the testing problem into two tasks and addressed them sequentially to find the optimal operation invocations and uncertainty values. Alternatively, one can try to reduce possible combinations of uncertainty values by, for instance choosing the minimum and maximum possible values of uncertainties or sample uncertainty values at a big interval. By taking such measures, a single-step algorithm might be devised. However, dedicated experiments are needed to compare our current approach with these alternatives.

In our empirical study, we applied seven policy-based reinforcement learning algorithms to find optimal uncertainty values. However, we acknowledge that evolutionary algorithms could also be applied together with reinforcement learning as demonstrated in (Khadka and Tumer 2018; Whiteson and Stone 2006). Due to limited resources, these algorithms were not included in this empirical study. It will be valuable to evaluate the performance of these algorithms in the future. Evolutionary algorithms can also be used alone to solve this testing problem.

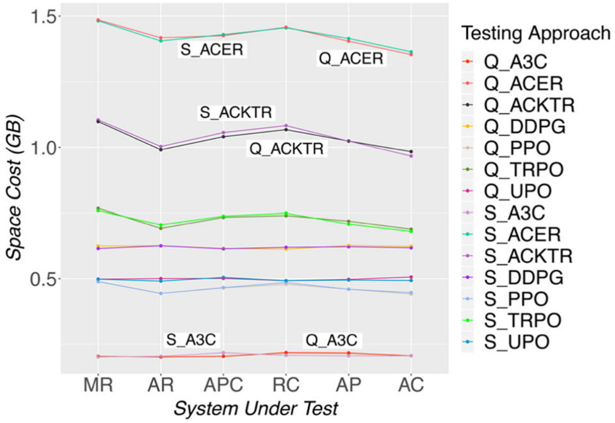

Fig. 10 Average Space Cost for Selecting and Introducing Uncertainties

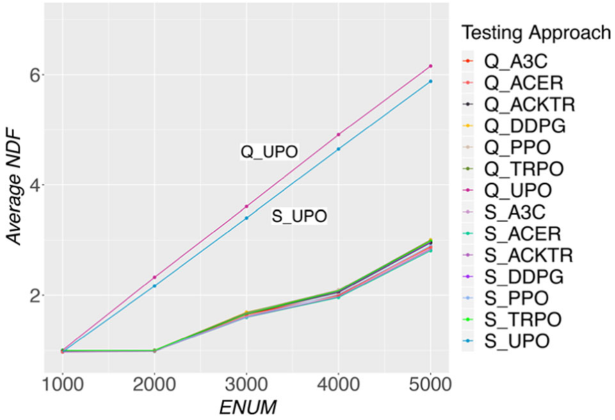

Fig. 11 Average Number of Detected Failures with Different ENUMs

However, to test a system based on state machines, evolutionary algorithms have to find valid transition paths first, which has already been proven as a challenging task (Derderian 2006; Lehre and Yao 2014). A walkaround solution is to generate all valid transition paths first, according to some coverage criteria, such as all transitions, and then apply evolutionary algorithms to select a subset of paths as a test suite (Lefticaru and Ipate 2007). Nevertheless, our previous experiment results demonstrate that covering all states and transitions is not sufficient for detecting failures in the SH-CPSs problem (Ma et al. 2019b). Consequently, we did not choose evolutionary algorithms to solve the testing problem.

A* is another popular algorithm that can be used to find the optimal path from a source to a destination, i.e., a transition path leading to the highest fragility in the context of our testing problem. However, to use the fragility as the heuristic of A* to find the optimal path, we have to know the fragility for each state. However, it is difficult (if not infeasible) to collect the fragilities for all states, since the number of possible states of an SH-CPS is huge. Moreover, each fragility has to be obtained from executions, which are computationally expensive and time-consuming. In contrast, reinforcement learning algorithms use an explore strategy (e.g., ε-greedy) to explore the space of all possible states. Guided by the estimated value function

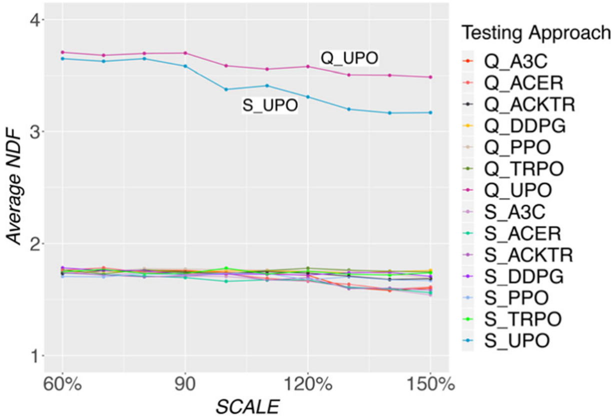

Fig. 12 Average Number of Detected Failures with Different SCALEs

(state value function or Q function), reinforcement learning algorithms can gradually find the path leading to the highest fragility.

Model checking is another approach that can be used to formally prove the correctness of a system. However, as we take the SH-CPS under test as a black box, it is unknown how the system's state variables' values are to be changed by an operation invocation or uncertainty value. Therefore, we could not use model checking to prove the correctness of an SH-CPS.

Due to these reasons, we only focused on evaluating the performance of different reinforcement learning algorithms in this empirical study. Fourteen combinations of reinforcement learning algorithms were applied to test six SH-CPSs, while more experiments are still needed to further address the threats to validity, explained in the next section.

7 Threats to Validity

This section analyzes the threats to validity from four aspects.

7.1 Construct validity

To evaluate the failure detection ability of the 14 combinations of reinforcement learning algorithms, we took the percentages of covered states (*SCov*) and transitions (*TCov*) and the number of detected failures (*NDF*) as the metrics. In addition, we further defined efficiency measure (*EFF*), time cost (*TCost*), and space cost (*SCost*), to investigate the efficiency and scalability of the algorithms. The metrics are comparable across the 14 combinations of algorithms, and they can directly reflect the effectiveness, efficiency, and cost of each combination.

One threat to construct validity is that the failures detected by the algorithms could be caused by potential flaws in test models rather than system defects. To mitigate the threats, first, we have defined four UML profiles to extend UML class diagrams and state machines (Ma et al. 2019a). Stereotypes defined in the profiles enable us to precisely specify expected functional behaviors, abnormal behaviors due to faults that occurred at runtime, self-healing behaviors for handling faults, and uncertainties that will affect these behaviors. Meanwhile, state invariants were used to define the valid ranges of state variables. The invariants enable us to rigorously define what behaviors are expected for a given state.

In addition to the above-mentioned rigorousness of the modeling notations, the modeling framework strictly enforces compliance with the UML standard and ensures syntactic correctness for the model. Moreover, we applied the framework to execute the models together with the SH-CPSs under test. As explained in Section 2.1, the framework could automatically compare the SH-CPSs' behaviors against the ones specified in the models. When a conflict was detected, we further examined whether this conflict was due to incorrectly specified models, including improper state invariants, wrong triggers or guards of transitions, and mismatched operations and testing interfaces. Consequently, we not only tested the SH-CPSs against the models, but also utilized the SH-CPSs to validate the models. In this way, we boosted the quality of the models and increased the credibility of the testing results.

7.2 Internal Validity

As explained in Section 2.3, we chose to test SH-CPSs in a two-steps approach, as it can reduce the search space an algorithm has to explore to find the optimal solution.

Nevertheless, additional experiments are still needed to verify if the two-steps approach is the best choice. Based on this two-steps approach, we evaluated the performance of 14 combinations of algorithms. The effectiveness and efficiency of a combination of reinforcement learning algorithms depend on the complexity of the system under test, the ranges of uncertainties that impact the system, the number of episodes the algorithms can take to detect failures, and hyperparameter settings of the algorithms.

In the experiment, we only compared the failure detection abilities of the algorithms for testing six subject systems with ten scales of uncertainty range and five settings of the number of episodes. The optimal combination of reinforcement learning algorithms found in this experiment may not perform the best in other settings, and thus more experiment results are needed to further confirm the conclusion.

Tuning the hyperparameters of the reinforcement learning algorithms is costly in terms of the time and computational devices that are required to conduct this task. Consequently, it is impractical and inefficient for testers to tune the hyperparameters every time before applying the algorithms to test a system. In this work, we only tuned the hyperparameters of each algorithm for three SH-CPSs with varying complexities. Although the tuned hyperparameters might not be the optimal one for all cases, they form a baseline and can be used as a starting point for future work.

7.3 Conclusion Validity

Due to the indeterminate policy used by the reinforcement learning algorithms to explore different operation invocations or uncertainty values, the number of detected failures and space/time costs are affected by randomness, threatening the conclusion validity. To reduce the threat, we repeated each testing job 10 times and applied statistical tests to evaluate the significance of the experiment result. We conducted the Kruskal-Wallis test (Kruskal and Wallis 1952) and Dunn's test (Dunn 1964) in conjunction with the Benjamini-Hochberg correction (Benjamini and Hochberg 1995) to check statistical significance, and Vargha and Delaney statistics (Vargha and Delaney 2000) to measure effect size. Finally, we acknowledge that more repetitions are needed to increase the trust on the results further.

7.4 External Validity

External validity concerns the generalization of the experiment results. In this experiment, we only tested three real-world systems, and three systems from the literature. They have 96 to 2160 states, and 270 to 2432 transitions. Each system is affected by a number of uncertainties, varying from 8 to 24. Although the results obtained from the six subject systems provide the evidence to support the conclusion, results from more SH-CPSs are still desired to validate the conclusion further.

8 Related Work

This section discusses related works on testing with reinforcement learning (Section 8.1) and testing under uncertainty (Section 8.2).

8.1 Testing with reinforcement learning

As a machine learning approach to solve sequential decision problems, reinforcement learning algorithms have been applied by a few researchers to solve several testing problems, as described below.

In a pioneering work, Veanes et al. devised an ad hoc reinforcement learning algorithm for online testing (Veanes et al. 2006). With the aim of covering more system behaviors, the algorithm keeps track of the number of times a transition has been triggered, and chooses a transition that has been triggered with the least of times. In their experiments, the ad hoc algorithm was compared with a random testing algorithm, and the proposed algorithms managed to cover more states than the random one, with much less time. However, the ad hoc algorithm does not consider the long-term reward, that is, the coverage of future transitions. Thus, the policy learned by this algorithm may be suboptimal.

In another work, Groce et al. proposed a light-weight automated testing framework for container-like classes (Groce et al. 2012). In their framework, SARSA (a value function learning algorithm, see Section 2.4.1) was used to learn the policy of generating test cases, i.e., sequences of method calls on container objects. In the evaluation, the SARSA based approach was compared with random testing and a modeling checking approach for 15 container classes. Their evaluation results show that the new approach performed better in 7 out of the 15 classes. As no other reinforcement learning algorithms were evaluated in the experiment, it is unknown whether other algorithms will perform better.

Mariani et al. and Reichstaller et al. applied Q-learning for GUI (Mariani et al. 2012) and interoperability testing (Reichstaller et al. 2016). In the first work, Q-learning was used to select the testing action that maximizes the changes of displayed GUI widgets, to cover functions of a system under test. In their empirical evaluation, the Q-learning based approach was compared with GUITAR, an open-source GUI testing tool, for four GUI applications. For all of these applications, the Q-learning based approach achieved a higher code coverage, and detected more faults than GUITAR. In the second work, Q-learning was applied to find implementation faults that can lead to the most critical failures, so that the riskiest implementation faults can be tested. The proposed testing approach was only evaluated by applying it for one case study, without comparing it with other methods.

Spieker et al. proposed a value function learning based reinforcement learning algorithm, similar to Q-learning, for test case prioritization (Spieker et al. 2017). In the evaluation, the reinforcement learning-based approach was compared with a random and two static test case prioritization approaches. Evaluation results demonstrated that the reinforcement learning-based approach could effectively learn to prioritize test cases that have a high chance to detect faults, with performance comparable with the two static methods, within 60 iterations.

More recently, Reichstaller and Knapp proposed a model-based reinforcement learning algorithm for testing self-adaptive systems (Reichstaller and Knapp 2018). Different from the reinforcement learning algorithms evaluated in this empirical study, the model-based algorithm tries to learn a Markov Decision Process (MDP) model of the system under test. An MDP model is defined by 1) a set of states of the system, 2) a set actions that can be performed on the system, 3) the transition probability that the system switches from one state to another when an action is conducted, and 4) the reward of performing an action under a state. When the MDP model is learned, it can be used to find the optimal policy of taking actions to maximize cumulative rewards. In the evaluation, the model-based algorithm was compared with Q-learning and a random method for testing a smart vacuum system. Testing results reveal that the model-based algorithm performed the best, and both model-based algorithm and Q-learning outperformed the random method. However, sufficient domain knowledge is needed to obtain the MDP model, and the current algorithm only supports learning the transition probability for a low-dimensional state space. These limitations restrict the applicability of the model-based reinforcement learning algorithm, and it needs further research to enhance the generalizability and learning capability.

In summary, existing works mainly evaluated the performance of value function learning based reinforcement learning algorithms for test case generation, prioritization, and risk-based testing. Besides, the hyperparameter settings used in these works, and how the hyperparameter settings were selected, were rarely mentioned in these papers. To find the optimal reinforcement learning algorithms for testing SH-CPSs under uncertainty, we conducted this empirical study and evaluated the performance of 14 combinations of reinforcement learning algorithms. By tuning these algorithms and applying them to test six SH-CPSs, we found the optimal reinforcement learning algorithms that detected the most failures in these systems, and with the least time cost.

8.2 Testing under Uncertainty

As uncertainty has been becoming prevalent in nowadays complex software systems, researchers have proposed approaches to either mitigate the uncertainty or test a system with uncertainties explicitly captured and introduced.

For uncertainty mitigation, Zhang et al. and Ji et al. both proposed to use Model-Based Testing (MBT) to discover unknown system behaviors due to indeterminate environmental conditions (Zhang et al. 2019) or uncertain networks (Ji et al. 2018). In another work (Camilli et al. 2020), Camilli et al. also applied MBT to collect actual system responses at runtime, and then the responses are fed to a Bayesian inference process that updates beliefs on uncertain parameters of system behaviors, modeled as a Markov Decision Process (MDP). Based on the result of the Bayesian inference process, values of the uncertain parameters are calibrated, and the calibrated MBP model can be used to support future development. From another perspective, Walkinshaw and Fraser proposed an uncertainty-driven Learning Based Testing (LBT) approach for unit testing (Walkinshaw and Fraser 2017). In this approach, Walkinshaw and Fraser apply genetic programming to learn multiple inference models of the program under test, based on previous testing results. Then, an active learning technique (Query By Committee) is used to select a test input, for which the inference models are the most uncertain about the outputs. The test input is then used to test the program, and the actual output of the program is used to further update the inference models, which are used to choose the next test input. Unlike these works that aim to discover unknown system behaviors or mitigate uncertainty, our work aims to find failures in SH-CPSs under a set of already identified uncertainties (measurement errors and actuation deviations), with the range of each uncertainty given.

To enable testing under uncertainty, Menghi et al. proposed an approach to generate test oracles for testing Simulink models with uncertain parameter values and white noises. In this approach, functional requirements are specified as Restricted Signals First Order Logic (RFOL) formulas and the formulas are transformed to Simulink blocks to calculate a quantitative measure, representing the degree of satisfaction of the requirements. Alternatively, in our work, we use a test model to capture the requirements of the system under test, and the constraints defined in the test model serve as test oracles. Simulation is also a common approach used to test systems under uncertainty. Ramirez et al. proposed to use simulators of sensors to test a goal model used by an adaptive system, with the measurement of sensors affected by noises and failures (Ramirez et al.

2011). Similarly, Minnerup and Knoll proposed to use simulators of actuators to test automated vehicles against a set of actuator inaccuracies (Minnerup and Knoll 2016). In these two works, the options of uncertainty are limited to either a few types, durations and severities of noises (Ramirez et al. 2011) or several samples of actuator inaccuracies, sampled from their ranges (Minnerup and Knoll 2016). On the contrary, our work aims to find a value within a valid range for each uncertainty and for each measurement or actuation, the uncertainty may take effect. Since the solution space of our testing problem is huge, we proposed to use reinforcement learning to effectively find the sequence of uncertainty values that can reveal a failure.

In summary, our work aims to find sequences of operation invocations and uncertainty values that make an SH-CPS failed to behave as expected, with the expected system behaviors captured as a test model and the range of each uncertainty given. This testing problem is different from the ones of the works mentioned above. We conducted this empirical study to find the optimal reinforcement learning algorithms for solving this test problem.

9 Conclusion

This paper presents an empirical study of applying reinforcement learning algorithms to test SH-CPSs under uncertainty, to find the optimal algorithms for failure detection. In this work, we applied 14 combinations of reinforcement learning algorithms to test six SH-CPSs, including two algorithms (State-Action-Reward-State-Action and Q-learning) for operation invocations, and seven algorithms (Asynchronous Advantage Actor-Critic, Actor-Critic method using Kronecker-factored Trust Region, Deep Deterministic Policy Gradient, Trust Region Policy Optimization, Proximal Policy Optimization, Actor-Critic method with Experience Replay, and Uncertainty Policy Optimization) for introducing uncertainties. Testing results reveal that the combination of Q-learning and Uncertainty Policy Optimization managed to detect the most failures, and on average, they took the least amount of time to detect a failure. Regarding the scalability of the algorithms, increasing the numbers of states and transitions of the system under test will incur extra space and time costs for SARSA and Q-learning, which were used for operation invocations. Whereas increasing the number of uncertainties has little effect on the costs of the other algorithms, which were used for introducing uncertainties.

Table 10 presents the statistical test results for the *Number of Detected Failures* (*NDF*). As APP_{Q_UPO} detected the most failures, we focus on comparing APP_{Q_UPO} with the other approaches. For each of the other 13 testing approaches, denoted as APP_c, we checked the p value of the Dunn's test, corresponding to the pair of APP_{Q_UPO} and APP_c. If the p value is over 0.05, the two testing approaches are considered to perform equally well, denoted as "=" in Table 10. Otherwise, we further computed the Vargha and Delaney statistics $\widehat{A}_{12}$, using the *NDF*s of the two approaches. If $\widehat{A}_{12}$ is above 0.5 for the pair of APP_{Q_UPO} and APP_c, it means APP_{Q_UPO} has a higher chance to detect more failures, and thus APP_{Q_UPO} is considered to be superior to APP_c, signified as ">". Otherwise, APP_{Q_UPO} is considered to be worse, signified as "<".

Table 11 summarizes the evaluation results for efficiency. We focus on comparing APP_{Q_UPO} with the other approaches. If an approach APP_c took significantly more (less) time than APP_{Q_UPO} to detect a failure, APP_c is inferior (superior) to APP_{Q_UPO} in terms of efficiency, denoted as ">" ("<") in Table 11.

Table 12 presents quantiles of time and space costs of each testing approach for the six SH-CPSs.

Appendix 1 Evaluation Results for Effectiveness

Table 10 Statistical Test Results for Effectiveness

SCALE	ENUM	$\mathrm{APP}_{\mathrm{Q\_UPO}}$ vs.																	
		$\mathrm{APP}_{\mathrm{Q\_A3C}}$			$\mathrm{APP}_{\mathrm{Q\_ACER}}$			$\mathrm{APP}_{\mathrm{Q\_PPO}}$			$\mathrm{APP}_{\mathrm{Q\_TRPO}}$			$\mathrm{APP}_{\mathrm{Q\_ACKTR}}$			$\mathrm{APP}_{\mathrm{Q\_DDPG}}$		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
60%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
70%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
80%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
90%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
100%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0

Table 10 (continued)

SCALE	ENUM	APP_{Q_UPO} vs.																	
		APP_{Q_A3C}			APP_{Q_ACER}			APP_{Q_PPO}			APP_{Q_TRPO}			APP_{Q_ACKTR}			APP_{Q_DDPG}		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
110%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
120%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
130%	1000	1	0	5	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
140%	1000	1	0	5	1	0	5	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
150%	1000	1	0	5	1	0	5	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	5	0	1
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
Sum		243	0	57	242	0	58	240	0	60	240	0	60	240	0	60	239	0	61
SCALE	ENUM	APP_{Q_UPO} **vs.**																	

Table 10 (continued)

SCALE	ENUM			APP_{Q_UPO} vs.																		
				APP_{Q_A3C}			APP_{Q_ACER}			APP_{Q_PPO}			APP_{Q_TRPO}			APP_{Q_ACKTR}			APP_{Q_DDPG}			
				>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>		<	=
		APP_{S_A3C}			APP_{S_ACER}			APP_{S_PPO}			APP_{S_TRPO}			APP_{S_ACKTR}			APP_{S_DDPG}			APP_{S_UPO}		
		>			<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>
<	=																					
60%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	1	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
70%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
80%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
90%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
100%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	1	0	5
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
110%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6

Table 10 (continued)

SCALE	ENUM			APP_{Q_UPO} vs.																		
				APP_{Q_A3C}			APP_{Q_ACER}			APP_{Q_PPO}			APP_{Q_TRPO}			APP_{Q_ACKTR}			APP_{Q_DDPG}			
				>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>		<	=
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
120%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
130%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	1	0	5
	2000	5	0	1	5	0	1	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	1	4
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
140%	1000	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6
	2000	4	0	2	4	0	2	6	0	0	6	0	0	6	0	0	6	0	0	0	0	6
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
150%	1000	1	0	5	0	0	6	0	0	6	0	0	6	1	0	5	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	5	0	1	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
SUM		238	0	62	236	0	64	240	0	60	240	0	60	241	0	59	240	0	60	19	2	279

Appendix 2 Evaluation Results for Efficiency

Table 11 Statistical Results for Efficiency

SCALE	ENUM	APP_{Q_UPO} vs.																	
		APP_{Q_A3C}			APP_{Q_ACER}			APP_{Q_PPO}			APP_{Q_TRPO}			APP_{Q_ACKTR}			APP_{Q_DDPG}		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
60%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
70%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
80%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
90%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	5	0	1	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
100%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0

Table 11 (continued)

SCALE	ENUM	$\mathrm{APP}_{\mathrm{Q\_UPO}}$ vs.																	
		$\mathrm{APP}_{\mathrm{Q\_A3C}}$			$\mathrm{APP}_{\mathrm{Q\_ACER}}$			$\mathrm{APP}_{\mathrm{Q\_PPO}}$			$\mathrm{APP}_{\mathrm{Q\_TRPO}}$			$\mathrm{APP}_{\mathrm{Q\_ACKTR}}$			$\mathrm{APP}_{\mathrm{Q\_DDPG}}$		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
110%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
120%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
130%	1000	1	0	5	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	5	0	1	5	0	1	5	1	0	5	0	1	5	0	1	5	0	1
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
140%	1000	0	0	6	1	0	5	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	5	0	1	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
150%	1000	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	6
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	5	0	1
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0

Table 11 (continued)

SCALE	ENUM	APP_{Q_UPO} vs.																	
		APP_{Q_A3C}			APP_{Q_ACER}			APP_{Q_PPO}			APP_{Q_TRPO}			APP_{Q_ACKTR}			APP_{Q_DDPG}		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
Sum		240	0	60	239	0	61	239	61	0	239	0	61	238	0	62	238	0	62

SCALE	ENUM	APP_{Q_UPO} vs.																				
		APP_{S_A3C}			APP_{S_ACER}			APP_{S_PPO}			APP_{S_TRPO}			APP_{S_ACKTR}			APP_{S_DDPG}			APP_{S_UPO}		
		>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>	<	=
60%	1000	2	0	4	2	0	4	2	0	4	1	0	5	2	0	4	2	0	4	1	0	5
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
70%	1000	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
80%	1000	1	0	5	1	0	5	1	0	5	1	0	5	1	0	5	1	0	5	1	0	5
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
90%	1000	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
100%	1000	1	0	5	1	0	5	2	0	4	1	0	5	1	0	5	1	0	5	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4

Table 11 (continued)

SCALE	ENUM			APP$_{Q\_UPO}$ vs.																			
				APP$_{Q\_A3C}$			APP$_{Q\_ACER}$			APP$_{Q\_PPO}$			APP$_{Q\_TRPO}$			APP$_{Q\_ACKTR}$			APP$_{Q\_DDPG}$			
				>	<	=	>	<	=	>	<	=	>	<	=	>	<	=	>		<	=
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
110%	1000	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	4	0	2	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
120%	1000	2	0	4	2	0	4	1	0	5	2	0	4	2	0	4	2	0	4	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0	5
130%	1000	3	0	3	2	0	4	2	0	4	3	0	3	2	0	4	2	0	4	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	1	2
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
140%	1000	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4	2	0	4
	2000	6	0	0	6	0	0	6	0	0	6	0	0	5	0	1	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
150%	1000	1	0	5	1	0	5	1	0	5	1	0	5	2	0	4	1	0	5	3	0	3
	2000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	3000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	2	0	4
	4000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
	5000	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	6	0	0	3	0	3
SUM		258	0	42	257	0	43	257	0	43	257	0	43	257	0	43	259	0	41	98	1	201

Appendix 3 Time and Space Costs of Reinforcement Learning Algorithms

Table 12 Time and Space Costs of Each Testing Approach for Six SUTs

APP	SUT	#States	#Transitions	#Unc.	Time Cost (s)			Space Cost (G)		
					1st Qu.	Median	3rd Qu.	1st Qu.	Median	3rd Qu.
Q_A3C	AC	432	432	24	15.6	25.3	39.0	5.7	6.8	7.6
	AR	140	360	12	6.9	9.2	11.6	5.3	5.7	6.1
	AP	96	270	24	6.1	7.9	10.6	4.6	5.0	5.9
	APC	1000	1008	18	22.6	37.1	56.2	6.6	7.8	9.2
	RC	2160	2432	18	52.6	65.4	87.0	13.2	15.3	17.0
	MR	1080	1656	8	35.6	46.2	63.1	8.2	9.2	9.9
Q_TRPO	AC	432	432	24	15.9	22.8	38.4	5.7	6.7	7.7
	AR	140	360	12	6.6	8.8	11.6	5.3	5.7	6.1
	AP	96	270	24	6.4	8.1	10.4	4.6	5.0	5.9
	APC	1000	1008	18	22.8	36.5	60.8	6.5	8.0	9.2
	RC	2160	2432	18	54.4	67.8	84.8	13.2	15.4	17.1
	MR	1080	1656	8	35.4	45.2	62.4	8.0	9.3	10.0
Q_UPO	AC	432	432	24	14.7	23.6	39.1	5.8	6.8	7.6
	AR	140	360	12	6.7	9.0	11.4	5.3	5.6	6.1
	AP	96	270	24	6.0	7.7	10.4	4.6	5.1	5.9
	APC	1000	1008	18	22.4	37.8	61.5	6.6	7.8	9.1
	RC	2160	2432	18	49.4	62.7	84.1	13.1	15.3	17.1
	MR	1080	1656	8	34.1	43.4	60.6	8.2	9.2	10.0
Q_PPO	AC	432	432	24	15.0	24.6	38.3	5.8	6.8	7.6
	AR	140	360	12	7.1	9.4	12.1	5.2	5.6	6.1
	AP	96	270	24	6.1	7.9	10.5	4.6	5.1	5.9
	APC	1000	1008	18	21.5	35.5	61.3	6.6	7.9	9.2
	RC	2160	2432	18	54.4	68.5	88.1	13.1	15.3	17.1
	MR	1080	1656	8	35.4	45.4	60.6	8.1	9.2	9.9
Q_DDPG	AC	432	432	24	15.3	22.4	37.7	5.8	6.8	7.6
	AR	140	360	12	6.8	9.1	11.8	5.3	5.7	6.1
	AP	96	270	24	6.2	7.8	10.8	4.6	5.0	5.9
	APC	1000	1008	18	21.3	33.6	59.8	6.5	8.0	9.2
	RC	2160	2432	18	52.8	67.2	85.6	13.2	15.3	17.0
	MR	1080	1656	8	34.5	43.5	58.7	8.2	9.2	9.9
S_ACKTR	AC	432	432	24	18.4	27.2	39.6	6.6	7.3	7.9
	AR	140	360	12	6.8	9.0	11.7	5.3	5.9	6.4
	AP	96	270	24	8.3	10.9	14.7	4.8	5.2	6.0
	APC	1000	1008	18	37.5	62.9	94.8	8.0	9.2	10.3
	RC	2160	2432	18	64.2	80.7	96.9	14.1	15.1	16.5
	MR	1080	1656	8	41.8	53.9	66.6	9.4	10.0	10.8
S_DDPG	AC	432	432	24	17.3	27.4	38.9	6.6	7.3	8.0
	AR	140	360	12	6.5	8.7	11.1	5.3	5.9	6.4
	AP	96	270	24	8.1	11.0	14.1	4.8	5.2	5.9
	APC	1000	1008	18	35.5	63.6	98.2	8.0	9.3	10.5
	RC	2160	2432	18	56.6	70.5	85.7	14.0	15.2	16.5
	MR	1080	1656	8	39.4	50.9	64.1	9.3	10.0	10.8
S_A3C	AC	432	432	24	19.5	28.2	39.2	6.6	7.3	7.9
	AR	140	360	12	6.3	8.3	10.8	5.3	5.8	6.4
	AP	96	270	24	8.0	10.4	13.9	4.8	5.2	5.9
	APC	1000	1008	18	30.7	52.3	88.6	8.0	9.2	10.6
	RC	2160	2432	18	57.4	69.9	85.5	13.9	15.1	16.5
	MR	1080	1656	8	39.1	51.1	65.0	9.3	9.9	10.9
S_ACER	AC	432	432	24	19.8	30.1	40.7	6.6	7.3	8.0

Table 12 (continued)

APP	SUT	#States	#Transitions	#Unc.	Time Cost (s)			Space Cost (G)		
					1st Qu.	Median	3rd Qu.	1st Qu.	Median	3rd Qu.
	AR	140	360	12	7.1	9.3	11.9	5.3	5.8	6.4
	AP	96	270	24	8.7	11.6	14.9	4.8	5.3	6.0
	APC	1000	1008	18	34.6	51.9	91.6	8.0	9.2	10.4
	RC	2160	2432	18	64.9	79.0	96.9	14.0	15.2	16.4
	MR	1080	1656	8	43.5	55.3	68.5	9.3	9.9	10.8
S_PPO	AC	432	432	24	18.0	26.9	38.3	6.6	7.3	7.9
	AR	140	360	12	6.3	8.6	11.1	5.3	5.8	6.4
	AP	96	270	24	7.8	10.4	13.6	4.8	5.2	6.0
	APC	1000	1008	18	34.5	57.7	91.8	8.0	9.3	10.4
	RC	2160	2432	18	58.4	72.3	88.4	13.9	15.1	16.4
	MR	1080	1656	8	39.3	50.7	64.4	9.3	10.0	10.8
S_TRPO	AC	432	432	24	19.1	27.3	38.8	6.7	7.3	8.0
	AR	140	360	12	6.3	8.5	10.9	5.3	5.8	6.4
	AP	96	270	24	7.8	10.9	14.1	4.8	5.2	6.0
	APC	1000	1008	18	31.0	54.8	95.9	8.0	9.3	10.5
	RC	2160	2432	18	57.6	69.0	89.0	14.0	15.3	16.4
	MR	1080	1656	8	40.2	51.0	65.4	9.3	9.9	10.8
Q_ACKTR	AC	432	432	24	16.2	22.9	35.0	5.8	6.8	7.6
	AR	140	360	12	7.0	9.1	11.6	5.3	5.7	6.1
	AP	96	270	24	6.6	8.4	10.8	4.6	5.0	5.9
	APC	1000	1008	18	23.8	39.6	67.7	6.6	7.9	9.3
	RC	2160	2432	18	59.0	72.3	94.4	13.1	15.3	17.0
	MR	1080	1656	8	36.5	45.1	63.5	8.2	9.3	9.9
S_UPO	AC	432	432	24	18.0	26.8	39.2	6.7	7.3	7.9
	AR	140	360	12	6.2	8.4	10.7	5.3	5.9	6.4
	AP	96	270	24	8.1	10.8	13.7	4.8	5.2	6.0
	APC	1000	1008	18	35.5	56.7	95.6	7.9	9.3	10.4
	RC	2160	2432	18	52.8	66.7	81.3	14.0	15.1	16.5
	MR	1080	1656	8	36.6	47.4	61.7	9.4	10.0	10.8
Q_ACER	AC	432	432	24	16.1	25.0	39.1	5.8	6.8	7.7
	AR	140	360	12	7.3	9.8	12.2	5.3	5.7	6.1
	AP	96	270	24	6.5	8.7	11.2	4.6	5.0	5.9
	APC	1000	1008	18	22.1	37.3	59.7	6.6	7.8	9.2
	RC	2160	2432	18	58.6	72.4	96.0	13.3	15.2	17.3
	MR	1080	1656	8	38.8	50.1	67.2	8.2	9.2	9.9

*#Unc: Number of uncertainty instances

Acknowledgements This work was supported by the Research Council of Norway funded MBT4CPS (grant no. 240013/O70) project. Tao Yue and Shaukat Ali are also supported by the Co-evolver project funded by the Research Council of Norway (grant no. 286898/LIS) under the category of Researcher Projects of the FRIPO funding scheme. Tao Yue is also supported by the National Nature Science Foundation of China (grant no. 61872182).

References

Ali S, Iqbal MZ, Arcuri A, Briand LC (2013) Generating test data from OCL constraints with search techniques. IEEE Trans Softw Eng 39(10):1376–1402

Arulkumaran K, Deisenroth MP, Brundage M, Bharath AA (2017) A brief survey of deep reinforcement learning. arXiv preprint arXiv:1708.05866

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: a practical and powerful approach to multiple testing. J R Stat Soc Ser B Methodol:289–300

Brandstotter M, Hofbaur MW, Steinbauer G, Wotawa F (2007) Model-based fault diagnosis and reconfiguration of robot drives, 2007 IEEE/RSJ international conference on intelligent robots and systems, vol 2007, San Diego, CA, USA, pp 1203–1209. https://doi.org/10.1109/IROS.2007.4399092

Camilli M, Gargantini A, Scandurra P (2020) Model-based hypothesis testing of uncertain software systems. Softw Test Verifi Reliab 30(2):e1730

Derderian KA (2006) Automated test sequence generation for finite state machines using genetic algorithms. Brunel University, School of Information Systems, Computing and Mathematics, Ph.D. thesis. http://bura.brunel.ac.uk/handle/2438/3062

Duan Y, Chen X, Houthooft R, Schulman J, Abbeel P (2016) Benchmarking Deep Reinforcement Learning for Continuous Control. Proceedings of The 33rd International Conference on Machine Learning, in PMLR 48: 1329–1338

Dunn OJ (1964) Multiple comparisons using rank sums. Technometrics 6(3):241–252

Groce A, Fern A, Pinto J, Bauer T, Alipour A, Erwig M et al (2012) Lightweight automated testing with adaptation-based programming, 2012 IEEE 23rd international symposium on software reliability engineering. Dallas, TX, USA 2012:161–170. https://doi.org/10.1109/ISSRE.2012.1

Güdemann M, Ortmeier F, Reif W (2006) Safety and dependability analysis of self-adaptive systems, second international symposium on leveraging applications of formal methods, verification and validation (isola 2006), vol 2006, Paphos, Cyprus, pp 177–184. https://doi.org/10.1109/ISoLA.2006.38

Henderson P, Islam R, Bachman P, Pineau J, Precup D, Meger D (2018) Deep reinforcement learning that matters. Proceedings of the AAAI Conference on Artificial Intelligence 32(1). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/11694

Jaderberg M, Dalibard V, Osindero S, Czarnecki WM, Donahue J, Razavi A, et al. (2017) Population based training of neural networks. arXiv preprint arXiv:1711.09846

Jedlitschka A, Ciolkowski M, Pfahl D (2008) Reporting experiments in software engineering. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to advanced empirical software engineering. Springer, London. https://doi.org/10.1007/978-1-84800-044-5_8

Ji R, Li Z, Chen S, Pan M, Zhang T, Ali S et al (2018) Uncovering unknown system behaviors in uncertain networks with model and search-based testing. In: IEEE 11th international conference on software testing, verification and validation (ICST), vol 2018, Västerås, Sweden, pp 204–214. https://doi.org/10.1109/ICST.2018.00029

Khadka S, Tumer K (2018) Evolution-guided policy gradient in reinforcement learning. arXiv preprint arXiv: 1611.01224

Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, El Emam K et al (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734

Kiumarsi B, Vamvoudakis KG, Modares H, Lewis FL (2017) Optimal and autonomous control using reinforcement learning: a survey. IEEE Trans Neural Netw Learn Syst 29(6):2042–2062

Kober J, Bagnell JA, Peters J (2013) Reinforcement learning in robotics: a survey. Int J Robot Res 32(11):1238–1274

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621

Lefticaru R, Ipate F (2007) Automatic state-based test generation using genetic algorithms, vol 2007. Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007), Timisoara, Romania, pp 188–195. https://doi.org/10.1109/SYNASC.2007.47

Lehre PK, Yao X (2014) Runtime analysis of the (1+ 1) EA on computing unique input output sequences. Inf Sci 259:510–531

Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, et al. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971

Ma T, Ali S, Yue T (2019a) Modeling foundations for executable model-based testing of self-healing cyber-physical systems. Softw Syst Model 18(5):2843–2873

Ma T, Ali S, Yue T, Elaasar M (2019b) Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach. Softw Qual J 27(2):615–649

Mariani L, Pezze M, Riganelli O, Santoro M (2012) AutoBlackTest: automatic black-box testing of interactive applications. In: 2012 IEEE fifth international conference on software testing, verification and validation, vol 2012, Montreal, QC, Canada, pp 81–90. https://doi.org/10.1109/ICST.2012.88

Martens J, Grosse R (2015) Optimizing neural networks with Kronecker-factored approximate curvature. Proceedings of the 32nd International Conference on Machine Learning, in Proceedings of Machine Learning Research 37:2408–2417. Available from http://proceedings.mlr.press/v37/martens15.html

Minnerup P, Knoll A (2016) Testing automated vehicles against actuator inaccuracies in a large state space. IFAC-PapersOnLine 49(15):38–43

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG et al (2015) Human-level control through deep reinforcement learning. Nature 518(7540):529

Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, Silver D, Kavukcuoglu K (2016) Asynchronous methods for deep reinforcement learning. In: Proceedings of the 33rd international conference on machine learning, vol 48. PMLR, pp 1928–1937

Munos R, Stepleton T, Harutyunyan A, Bellemare M (2016) Safe and efficient off-policy reinforcement learning. arXiv preprint arXiv:1606.02647

OMG Specification (2006) Object constraint language V2.4. OMG file id: formal/14–02-03, https://www.omg.org/spec/OCL/2.4/PDF

OMG Specification (2016) Semantics of a foundational subset for executable UML models V1.2.1. OMG file id: formal/2016-01-05. https://www.omg.org/spec/FUML/1.1/PDF

OMG Specification (2017) Precise semantics of UML State Machines (PSSM). V1.0. OMG file id: formal/19–05-01. https://www.omg.org/spec/PSSM/1.0/PDF

Pascanu R, Bengio Y (2013) Revisiting natural gradient for deep networks. arXiv preprint arXiv:1301.3584

Pollard D (2000) Asymptopia: an exposition of statistical asymptotic theory. Available at http://www.stat.yale.edu/~pollard/Books/Asymptopia

Priesterjahn C, Steenken D, Tichy M (2013) Timed Hazard analysis of self-healing systems. In: Cámara J, de Lemos R, Ghezzi C, Lopes A (eds) Assurances for self-adaptive systems. Lecture notes in computer science, vol 7740. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-36249-1_5

Ramirez AJ, Jensen AC, Cheng BH, Knoester DB (2011) Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: 2011 26th IEEE/ACM international conference on automated software engineering (ASE 2011). Lawrence, KS, USA, pp 568–571. https://doi.org/10.1109/ASE.2011.6100127

Reichstaller A, Knapp A (2018) Risk-based testing of self-adaptive systems using run-time predictions. In: 2018 IEEE 12th international conference on self-adaptive and self-organizing systems (SASO), vol 2018, Trento, Italy, pp 80–89. https://doi.org/10.1109/SASO.2018.00019

Reichstaller A, Eberhardinger B, Knapp A, Reif W, Gehlen M (2016) Risk-based interoperability testing using reinforcement learning. In: Wotawa F, Nica M, Kushik N (eds) Testing software and systems. ICTSS 2016. Lecture notes in computer science, vol 9976. Springer, Cham. https://doi.org/10.1007/978-3-319-47443-4_4

Royston P (1995) Remark AS R94: a remark on algorithm AS 181: the W-test for normality. J R Stat Soc: Ser C: Appl Stat 44(4):547–551

Schulman J, Levine S, Abbeel P, Jordan M, Moritz P (2015) Trust region policy optimization. Proceedings of the 32nd International Conference on Machine Learning, in Proceedings of Machine Learning Research 37: 1889–1897. Available from http://proceedings.mlr.press/v37/schulman15.html

Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal policy optimization algorithms. *arXiv* preprint arXiv:1707.06347

Singh S, Jaakkola T, Littman ML, Szepesvári C (2000) Convergence results for single-step on-policy reinforcement-learning algorithms. Mach Learn 38(3):287–308

Spieker H, Gotlieb A, Marijan D, Mossige M (2017) Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, pp 12–22. https://doi.org/10.1145/3092703.3092709

Steinbauer G, Mörth M, Wotawa F (2006) Real-time diagnosis and repair of faults of robot control software. In: Bredenfeld A, Jacoff A, Noda I, Takahashi Y (eds) RoboCup 2005: robot soccer world cup IX. RoboCup

2005. Lecture notes in computer science, vol 4020. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11780519_2

Sutton RS, Barto AG (1998) Reinforcement learning: An introduction (Vol. 1, Vol. 1). MIT press, Cambridge

Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132

Veanes M, Roy P, Campbell C (2006) Online testing with reinforcement learning. In: Havelund K, Núñez M, Roşu G, Wolff B (eds) Formal approaches to software testing and runtime verification. FATES 2006, RV 2006. Lecture notes in computer science, vol 4262. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11940197_16

Venkatasubramanian V, Rengaswamy R, Yin K, Kavuri SN (2003) A review of process fault detection and diagnosis: part I: quantitative model-based methods. Comput Chem Eng 27(3):293–311

Walkinshaw N, Fraser G (2017) Uncertainty-driven black-box test data generation. In: 2017 IEEE international conference on software testing, verification and validation (ICST), Tokyo, Japan, 2017, pp. 253–263. https://doi.org/10.1109/ICST.2017.30

Wang Z, Bapst V, Heess N, Mnih V, Munos R, Kavukcuoglu K, et al. (2016) Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224

Whiteson S, Stone P (2006) Evolutionary function approximation for reinforcement learning. J Mach Learn Res 7(May):877–917

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. IEEE Trans Softw Eng SE-12(7):733–743, July 1986. https://doi.org/10.1109/TSE.1986.6312975

Wu Y, Mansimov E, Grosse RB, Liao S, Ba J (2017) Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. arXiv preprint arXiv:1708.05144

Zhang M, Ali S, Yue T (2019) Uncertainty-wise test case generation and minimization for cyber-physical systems. J Syst Softw 153 Elsevier. https://doi.org/10.1016/j.jss.2019.03.011

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Tao Ma is currently a Ph.D. student in Simula Research Laboratory and University of Oslo (Norway). His current research interests mainly focus on devising automated testing solutions for large-scale intelligent systems such as self-adaptive systems and cyber-physical systems. His expertise includes model execution and machine learning (particularly supervised learning and reinforcement learning).

Shaukat Ali is a chief research scientist at Simula Research Laboratory, Oslo, Norway and the Head of the Department of Engineering Complex Software Systems. His research focuses on devising novel methods for Verification and Validation (V&V) of large scale highly connected software-based systems. He has been involved in a number of basic research, research-based innovation, and innovation projects in the capacity of PI/Co-PI related to Model-based Testing (MBT), Search-Based Software Engineering, Model-Based System Engineering, and Quantum Software Testing. He has rich experience of working in several countries including UK, Canada, Norway, Japan and Pakistan.

Tao Yue is a full professor at Nanjing university of Aeronautics and Astronautics, also an adjunct scientist to Simula Research Laboratory. She has more than 20 years of experience of conducting industry-oriented research with a focus on Model-Based Engineering (MBE) in various application domains such as Maritime and Energy, Communications, Automated Industry, and Healthcare in several countries including Canada, Norway, and China. Her present research area is software engineering, with specific interests in MBE, Model-based Testing, Uncertainty-wise Testing, Uncertainty Modeling, Search-based Software Engineering, Empirical Software Engineering, Product Line Engineering, and Requirements Engineering, Quantum Software Engineering.

Affiliations

Tao Ma[1,2] **· Shaukat Ali**[1] **· Tao Yue**[1,3]

Tao Ma
taoma@simula.no

Shaukat Ali
shaukat@simula.no

[1] Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

[2] University of Oslo, P.O. Box 1072, 0316 Blindern, Oslo, Norway

[3] Nanjing University of Aeronautics and Astronautics, Nanjing, China