



Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach

Tao Ma^{1,2} · Shaukat Ali¹ · Tao Yue¹ · Maged Elaasar³

Published online: 20 March 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

As an essential feature of smart cyber-physical systems (CPSs), self-healing behaviors play a major role in maintaining the normality of CPSs in the presence of faults and uncertainties. It is important to test whether self-healing behaviors can correctly heal faults under uncertainties to ensure their reliability. However, the autonomy of self-healing behaviors and impact of uncertainties make it challenging to conduct such testing. To this end, we devise a fragility-oriented testing approach, which is comprised of two novel algorithms: fragility-oriented testing (FOT) and uncertainty policy optimization (UPO). The two algorithms utilize the fragility, obtained from test executions, to learn the optimal policies for invoking operations and introducing uncertainties, respectively, to effectively detect faults. We evaluated their performance by comparing them against a coverage-oriented testing (COT) algorithm and a random uncertainty generation method (R). The evaluation results showed that the fault detection ability of FOT+UPO was significantly higher than the ones of FOT+R, COT+UPO, and COT+R, in 73 out of 81 cases. In the 73 cases, FOT+UPO detected more than 70% of faults, while the others detected 17% of faults, at the most.

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

✉ Tao Yue
tao@simula.no

Tao Ma
taoma@simula.no

Shaukat Ali
shaukat@simula.no

Maged Elaasar
melaasar@gmail.com

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

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

³ Carleton University, 1125 Colonel By Dr., Ottawa, ON K1S5B6, Canada

1 Introduction

The integration of computation, communication, and control awards cyber-physical systems (CPSs) with a higher level of intelligence, which enables them to autonomously adapt and optimize their behavior at runtime (Bures et al. 2015). One of such autonomous characteristics is self-healing, which endows CPSs with the ability to detect fault occurrences, diagnose causes, and recover. We refer to this kind of CPSs as self-healing CPSs (SH-CPSs).

Besides recuperation, the self-healing behaviors of SH-CPSs have to deal with uncertainty gracefully. Due to intimate coupling between the cyber and physical components, SH-CPSs are usually affected by various uncertainties. By uncertainty, we mean “the lack of knowledge of which value an uncertain factor will take at a given point in time during execution” (Ma et al. 2018). In this paper, we limit our scope to environmental uncertainty, namely, measurement uncertainties from sensors and actuation deviations from actuators.

Since self-healing behaviors play a key role in securing CPSs’ normal functionality, it is important to check the correctness of self-healing behaviors in the presence of uncertainty. However, achieving this task is non-trivial. Although formal verification can rigorously prove the correctness, these technologies are still not applicable to large-scale applications, due to high computational complexity particularly when many uncertainties need to be considered (Schupp et al. 2015). Testing is another option. However, at the current stage, the state of practice of testing CPSs is an ad hoc, trial and error testing approach, which cannot provide sufficient rigor in fault detection (Zheng and Julien 2015). For the state of art of testing CPSs, coverage-oriented structural testing is dominating (Asadollah et al. 2015). However, the high dimension of CPSs’ behaviors, the tight integration of cyber and physical components, and the unpredictable operational environment make the space of CPSs’ behaviors extremely large. It is difficult to find faults in such huge space by just randomly searching or trying out each possibility. Note that there are two kinds of *faults*: flaws in the SH-CPS under test (SUT) and faults targeted by self-healing behaviors. Except particular explanation, a *fault* in the following paragraphs refers to a flaw instead of a *fault* injected for testing self-healing behaviors.

To overcome the limitations of existing methods, we propose a fragility-oriented approach. In this approach, we try to identify how likely the SUT is going to fail in a given state, i.e., the *fragility* of the SUT. The fragility is used as a heuristic to guide the testing process to spend more testing effort on the fragile states so that faults can be more effectively detected.

To detect faults in the SUT under uncertainty, we have to apply fragility to select two kinds of inputs. The first is operation invocation, which controls the behavior performed by the SUT. Invoking different operations or calling the same set of operations with different orders may both lead to distinct system states. The second is uncertainty values. They define the uncertainty-introduced environment, where the SUT is executed. Both operation invocations and uncertainty values need to be cautiously selected to find the most fragile state and detect faults.

For operation invocation, we have devised a fragility-oriented testing (FOT) algorithm. It employs a reinforcement learning approach to find the optimal sequence of operation invocations concerning fault revelation. Regarding the generation of uncertainty values, we proposed a distribution-based generation method in our previous work (Ma et al. 2017). In this method, the variation of each uncertainty is expressed as a probability or possibility distribution. Based on the distribution, uncertainty values are generated. Since this method merely derives uncertainty values from fixed distributions, without utilizing any heuristic, it is suboptimal regarding the effectiveness of generating uncertainties for fault revelation.

To overcome this weakness, we present an uncertainty policy optimization (UPO) algorithm in this paper. The algorithm uses a parameterized policy to address the uncertainty generation problem. The policy takes state variable values of the SUT as input. Based on the values, the policy decides the uncertainty values that should be introduced for the current state to increase the fragility of the SUT. Directed by the fragility obtained from executions, the UPO algorithm gradually optimizes the policy in terms of the fragility of the SUT that can be achieved by following the policy. In such way, the UPO algorithm manages to effectively find a sequence of uncertainty values that can work together with a sequence of operation invocations to reveal a fault.

We compared the performance of UPO and FOT against coverage-oriented testing (COT) (Veanes et al. 2006) and the random uncertainty generation method (R) by applying them to test nine self-healing behaviors of three real-world case studies. Each self-healing behavior was tested under eight uncertainties, with three settings of time budgets and three ranges of uncertainty variation. In total, 81 testing jobs (nine self-healing behaviors \times three testing times \times three uncertainty scales) were accomplished by each testing approach. The experiment results showed that the fault detection ability of FOT+UPO is significantly higher than the ones of FOT+R, COT+UPO, and COT+R, in 73 out of 81 cases. In the 73 cases, FOT+UPO detected more than 70% of faults, while merely less than 17% of faults were detected by the other three approaches.

This paper is an extension of our previous conference paper (Ma et al. 2017). The new contributions of this paper are as follows: (1) the fragility-oriented testing (FOT) algorithm has been improved with the ability to detect multiple faults. (2) A new fragility-oriented algorithm—uncertainty policy optimization (UPO) has been devised for uncertainty generation. (3) The performances of the two algorithms have been evaluated by comparing the numbers and percentages of detected faults of four testing approaches, using nine self-healing behaviors from three case studies.

We organize the paper as follows. Section 2 presents the background, followed by a running example given in Section 3. Section 4 presents an overview of the fragility-oriented testing approach. Section 5 and Section 6 present the FOT and UPO algorithms, respectively. Section 7 illustrates the implementation. Section 8 presents the evaluation, Section 9 summarizes related work, and Section 10 concludes the paper.

2 Background

The proposed fragility-oriented testing approach is devised based on two fundamental techniques. One is model execution and the other is reinforcement learning. This section introduces two kinds of models used in the approach—Executable Test Model (ETM) and Dynamic Flat State Machine (DFSM) in Section 2.1 and Section 2.2, respectively. Section 2.3 summarizes a test model execution framework—TM-Executor. Section 2.4 describes the general idea of reinforcement learning and Section 2.5 explains how to use an artificial neural network (ANN) to facilitate reinforcement learning.

2.1 Executable Test Model

A CPS can be seen as a set of networked physical units, working together to monitor and control physical processes. A physical unit can be further decomposed into sensors, actuators,

and controllers. A controller monitors and controls physical processes via sensors and actuators, which are functional behaviors. As a specific type of CPSs, SH-CPSs can monitor fault occurrences and adapt its behavior to self-healing behaviors when a fault occurs. As the objective of a self-healing behavior is to restore functional behaviors, both *expected* functional and self-healing behaviors need to be captured for testing. Previously, we proposed a UML-based modeling framework, called MoSH (Ma et al. 2018), which allows creating an ETM for the SH-CPS under test (SUT). The ETM consists of a set of UML state machines annotated with dedicated stereotypes from the MoSH profiles.

The set of state machines captures expected functional and self-healing behaviors of the SUT: $SM = \{sm_1, \dots, sm_i, \dots, sm_n\}$, where MoSH stereotypes are applied to annotate the states in state machines. A sm_i has a set of states $S_{sm_i} = \{s_{sm_i1}, \dots, s_{sm_ij}, \dots, s_{sm_iss}\}$ and transitions $T_{sm_i} = \{t_{sm_i1}, \dots, t_{sm_ik}, \dots, t_{sm_it}\}$. A state s_{sm_ij} ($s_{sm_ij} \in S_{sm_i}$) is defined by a *state invariant* O_{sm_ij} , which is specified as an OCL¹ constraint, constraining one or more state variables. When s_{sm_ij} is active, its corresponding state invariant has to be satisfied. A transition t_{sm_ik} ($t_{sm_ik} \in T_{sm_i}$) is defined as a tuple $t := (s_{src}, s_{tar}, op, g)$, where s_{src} and s_{tar} are the source and target states of t , op denotes an operation call event that can trigger the transition,² and the operation represents a testing interface used to control the SUT. g signifies the transition's guard, an OCL constraint. It restricts input parameter values that can be used to invoke the operation for firing the transition. By conforming to the fUML³ and Precise Semantics Of UML State Machines (PSSM)⁴ standards, the specified state machines are executable. Thus, the test model is called an ETM.

2.2 Dynamic Flat State Machine

Test execution with concurrent and hierarchical state machines is computationally expensive and complex. Since statically flattening state machines may lead to state explosion, we implemented an algorithm to dynamically and incrementally flatten UML state machines into a Dynamic Flat State Machine (DFSM) during test execution. A DFSM has a set of states $\mathbb{S} = \{\mathbb{s}_1, \mathbb{s}_2, \dots, \mathbb{s}_\alpha, \dots, \mathbb{s}_\kappa\}$ and a set of transitions $\mathbb{T} = \{\mathbb{t}_1, \mathbb{t}_2, \dots, \mathbb{t}_\beta, \dots, \mathbb{t}_m\}$. Each state \mathbb{s}_α in \mathbb{S} is constituted by states from each sm_i , denoted as $\mathbb{s}_\alpha = s_{sm_1x} \wedge s_{sm_2y} \wedge \dots \wedge s_{sm_nz}$. Accordingly, the conjunction of all constituents' state invariants $\left[o_{sm_1x'} \wedge o_{sm_2y'} \wedge \dots \wedge o_{sm_nz'} \right]$ forms the state invariant of \mathbb{s}_α , denoted as $\mathbb{o}_{\alpha'}$. Meanwhile, the set of transitions connecting the DFSM states is captured by \mathbb{T} . In the test model, the interactions among different state machines are modeled by transitions with effects of sending signals (Ma et al. 2018). When such a transition is triggered, it sends signals that activate the transitions in other state machines. The set of activated transitions are represented by the initially triggered transition in the flattened state machine. Consequently, each transition \mathbb{t}_β belonging to \mathbb{T} is uniquely mapped to a transition t_{sm_xt} in a state machine sm_x , expressed as $\mathbb{t}_\beta = t_{sm_xt}$. While the Executable Test Model (ETM) is being executed, the DFSM is dynamically constructed. The fragility-oriented testing (FOT) algorithm uses the DFSM to learn the value of firing each transition and find the optimal

¹ <http://www.omg.org/spec/OCL/2.4>

² Though call, change, and signal event occurrences can all be triggers to model expected behaviors, only transitions having call event occurrences as triggers can be activated from the outside. A change event or a signal event is only for the SUT's internal behaviors, which cannot be controlled for testing.

³ <http://www.omg.org/spec/FUML/1.2.1>

⁴ <http://www.omg.org/spec/PSSM/1.0/Beta1>

transition selection policy to effectively find faults. Thus, we mainly use DFSM in the following paragraphs.

2.3 Test model execution framework

We developed a testing framework called TM-Executor in our previous work (Ma et al. 2018). By executing the test model and the SUT at the same time, the framework can dynamically test the SUT against the model. Figure 1 presents the execution process. According to the execution semantics of UML state machines, TM-Executor executes the test model, i.e., a set of UML state machines (S1). During the execution, TM-Executor dynamically and incrementally derives a DFSM from the set of state machines (S2). The DFSM points out the candidate transitions that can be triggered to drive the execution of the model. Directed by an operation invocation policy, TM-Executor selects a transition and generates an operation invocation to trigger the transition (S3~S7). As aforementioned, a transition’s trigger *op* and guard *g* specify the operation and the parameter values to be used to trigger the transition. While an operation is invoked, an operation call event is generated, which drives the execution of the test model. Meanwhile, the operation is executed to call a corresponding testing interface, which makes the SUT enter the next state.

Two kinds of testing interfaces can be specified as a transition’s trigger *op*. One is functional control operation, which instructs the SUT to execute a nominal functional operation. Another is fault injection operation, which introduces a fault in the SUT, based on which TM-Executor controls when and which faults to be injected to the SUT to trigger its self-healing behaviors.

On the other hand, TM-Executor uses an uncertainty generation policy to generate the uncertainty values and introduces the uncertainty values into the SUT to test the system under uncertainty (S8~S10). Via testing interfaces, state variable values are queried from the SUT and used by TM-Executor to evaluate state invariants of the active state (S11). If an invariant is evaluated false, it means that the SUT fails to behave consistently with the ETM and a fault is detected.

2.4 Reinforcement learning

To effectively detect faults in SH-CPSs under uncertainty, we aim to find the optimal policy for invoking operations and introducing uncertainties. The policy helps us find the sequence of operation invocations together with the sequence of uncertainty values that can reveal faults. Finding such optimal policies is exactly the goal of reinforcement learning, an automatic

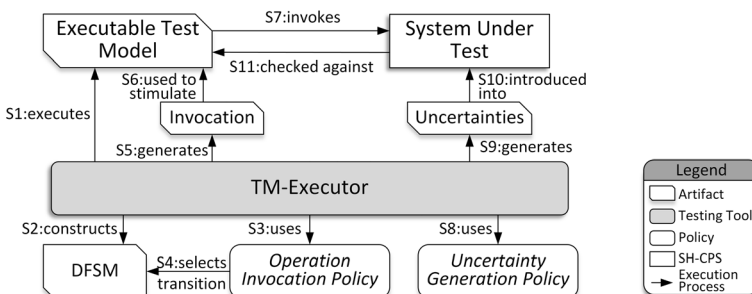


Fig. 1 Test execution process

approach to learning the optimal policy from interactions (Sutton and Barto 1998). Consequently, we devise two reinforcement learning–based algorithms to facilitate testing SH-CPSs under uncertainty.

Figure 2 presents the general idea of reinforcement learning in the context of testing. The reinforcement-learning algorithm directs a testing agent to take testing actions on the SUT, with the aim of maximizing the possibility for the SUT to fail, i.e., maximizing the likelihood to detect faults. The agent tests the SUT in discrete time steps. At each time step t , the agent uses testing interfaces to obtain the state of the SUT S_t , represented as a collection of state variables. After that, it selects a testing action A_t from the set of available actions in state S_t . Caused by the action, the state of the SUT changes from S_t to S_{t+1} . The agent evaluates F_{t+1} —the likelihood that the SUT is to fail in S_{t+1} , which is defined as fragility in this paper. Taking the fragility as a heuristic, the reinforcement algorithm continuously adjusts the agent’s action selection policy to achieve the highest fragility and effectively detect faults.

2.5 Artificial neural network

The policy used in the reinforcement learning can be saved in two ways. One is tabular form, which explicitly specifies the probability to take action in a given state. However, when the number of potential states or the number of valid actions becomes huge, it is intractable to store the probability for each pair of state and action. In this case, the policy has to be stored in an approximate form. One well-known form is artificial neural network (ANN) (Yegnanarayana 2009), which has been successfully applied together with reinforcement learning in many algorithms (Arulkumaran et al. 2017; Li 2017).

An ANN consists of layers of interconnected neurons, as shown in Fig. 3. The first layer is an input layer. Each neuron in the input layer represents one dimension of the input space, and the activity of these neurons is just outputting the value of the corresponding dimension. Via weighted connections, the value is scaled by the weights of connections and transited to the neurons in the next layer, which is called the hidden layer. The neurons in the hidden layer called hidden neurons add a bias to the sum of received values. After that, they input the result to an activation function and send the output of this function to all their successors. The values pass through the network in this way until reaching the last layer, the output layer. The neurons in the output layer are called output neurons. The activity of output neurons is the same with hidden neurons, except that the output neurons use an output function instead of the activation function to calculate the final outputs.

Via the network structure, the ANN can compactly save the mapping relations between inputs and outputs. In the context of reinforcement learning–based testing, the input is the state of the SUT, and the output is the testing action to be performed in that state. However, this benefit of applying ANN is at the cost of lower accuracy. Since it is almost infeasible to train

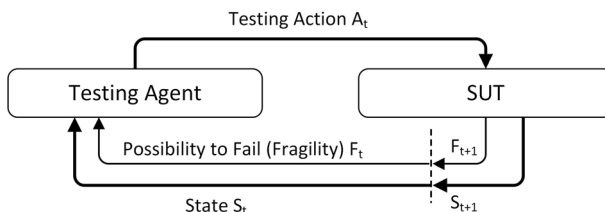


Fig. 2 Testing with reinforcement learning (Sutton and Barto 1998)

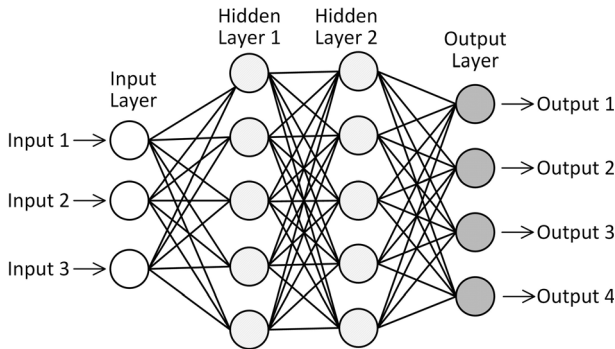


Fig. 3 Example of artificial neural network

an ANN with 100% accuracy (Yegnanarayana 2009), an estimation error will be introduced when the ANN instead of a tabular form is used to estimate the output for a given input. Also, training an ANN is computationally expensive. Compared with tabular form, applying ANN to save the policy for reinforcement learning requires more computational resources, and it takes an extra amount of time to train the ANN (Sutton and Barto 1998).

In this paper, we divide the testing task into two subtasks. One is responsible for selecting operation invocations, and the other takes charge of introducing uncertainties. For the first subtask, the test model specifies the valid operation invocations for each state. When introducing uncertainties, each uncertainty can arbitrarily take any value within its variation range. Therefore, the search space of the second subtask is significantly larger than the one of the first subtask. Due to this reason, we devise a tabular form-based algorithm for the first subtask and apply ANN to address the second subtask.

3 Running example

We use an unmanned aerial vehicle control system (i.e., ArduCopter⁵) as a running example to illustrate the problem of testing SH-CPS under uncertainty. Figure 4 presents a UML class diagram, which captures a simplified architecture of the system. In the diagram, each class represents a sensor, actuator, controller, or physical unit; accessible state variables are specified as class attributes; and the operations capture available testing interfaces.

As shown in Fig. 4, ArduCopter has two physical units, i.e., *Copter* and *Ground Control Station* (GCS). With the GCS, users remotely control the *Copter* using some flight modes. During the flight, the *Copter* is constantly affected by environmental uncertainties such as measurement bias from the GPS. The uncertainties are specified via the «Uncertainty» stereotype, provided in MoSH profile (Ma et al. 2018). An example is shown in the upper right corner of Fig. 4. The stereotype attribute *universe* specifies the variation range of the uncertainty, and *measure* defines the uncertainty's probability distribution. For the uncertainty *posBias*, i.e., the measurement bias of position, its variation range is between -2.5 and 2.5 , and the value of the uncertainty follows a normal distribution with mean 0 and variance 0.9. These are specified based on the product specification of the GPS.

⁵ <http://ardupilot.org/copter/>

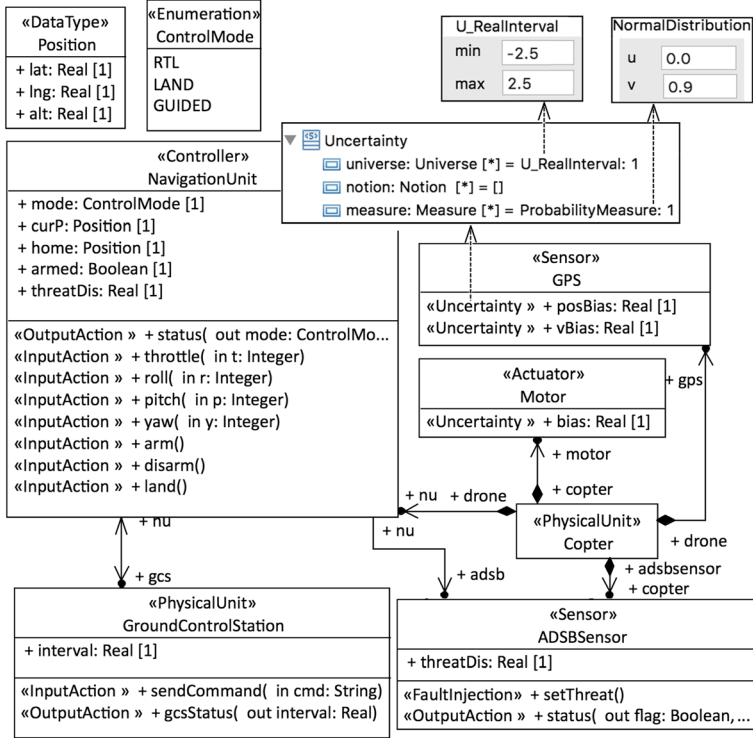


Fig. 4 Simplified architecture of ArduCopter

Based on the class diagram, the expected behaviors of the classes are specified as an ETM (shown in Fig. 11 in Appendix). The ETM captures both functional and self-healing behaviors. The functional behaviors, such as *FlightControlBehavior* and *ADSBBehavior*, specify how the system should behave when an operation is invoked, and the self-healing behaviors specify how a fault is to be healed.

CollisionAvoidance is one of the self-healing behaviors. Due to improper flight control (operational fault), the copter may approach another aircraft. In such case, the copter automatically adapts the velocity and orientation (i.e., the angles of rotations in roll, pitch, and yaw) of the flight to avoid a collision.

Figure 5 presents a partial simplified DFSM corresponding to the ETM for ArduCopter. We take one path (bold transitions in Fig. 4: $t1 \rightarrow t2 \rightarrow t3 \rightarrow t4 \rightarrow t11 \rightarrow t12 \rightarrow t19 \rightarrow t21$) to explain test execution.

Starting from the *Initial* state, the DFSM directly enters *Stopped*, as no trigger is required to enter the first state. From *Stopped*, TM-Executor fires $t2$ by calling the functional control operation *arm* to launch the *Copter*. As a result, *Started* becomes active. To make the system enter state *Lift*, TM-Executor invokes operation *throttle* with a valid value of input parameter t obtained by solving guard constraint [$t > 1500$ and $t < 2000$] via constraint solver EsOCL (Ali et al. 2013). Then, the *Copter* takes off and reaches the *Lift* state. In the *Lift* state, TM-Executor invokes *throttle* with $t = 1500$. This invocation triggers the *Copter* to hover above the ground. In the *Hovering* state, TM-Executor either changes the *Copter*'s movement (i.e., firing $t5$, $t7$, or $t19$) or invokes the fault injection operation *setThreat*, which simulates that an aircraft is approaching from the left behind of

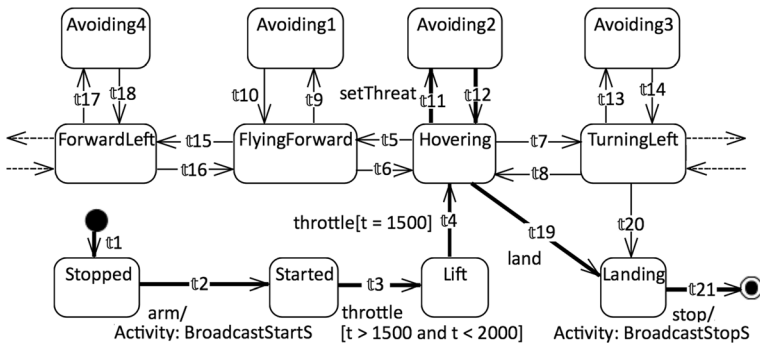


Fig. 5 Simplified partial DFSM for ArduCopter

the *Copter* to trigger the collision avoidance behavior. Assume the second option is adopted. Triggered by this, the collision avoidance behavior controls the *Copter* to fly away from the approaching aircraft. When the distance between them (*threatDis*) is over 1000 m (not shown in Fig. 5), the collision threat is avoided and the *Copter*'s flight mode changes back to the previous one. Hence, t_{12} is traversed.⁶ Then TM-Executor chooses to trigger t_{19} , followed by firing t_{21} , to stimulate the copter to pass through the *Landing* state and reaches the final state.

In addition to operation invocations, a sequence of uncertainty values is required to execute ArduCopter under uncertainty. Since the control loop frequency of the copter is 400 Hz, the copter's controller reads sensor data and outputs actuation commands every 2.5 ms. Each reading and controlling is potentially affected by uncertainties like measurement noise from the GPS and actuation deviation from the motor. Therefore, every 2.5 ms, the value of each uncertainty has to be generated and be used to impact the copter's sensing or actuating to simulate the effect of uncertainties.

In parallel to the execution, TM-Executor periodically obtains the values of the SUT's state variables through testing interfaces and repeatedly uses these values to evaluate the active state's invariant, using the constraint evaluator Dresden OCL (Demuth and Wilke 2009). If an invariant is evaluated to be false, then a fault is detected.

The decisions of which operation to invoke and which uncertainty values to use determine whether a fault can be found in an execution. From specifications, we know that there is a fault in the collision avoidance behavior when an aircraft is approaching from -45° and the copter is flying to the forward left; the collision avoidance behavior has to reverse the copter's orientation to make the two aerial vehicles fly away. Since reversing the orientation takes more time than other orientation adjustments, the copter, in this case, flies closer to the approaching aircraft. Due to noisy sensor data and inaccurate actuations, a collision does have a chance to occur in this condition.

To detect the fault leading to the collision, the fault injection operation *setThreat* needs to be invoked in state *ForwardLeft*, i.e., t_{17} must be activated. However, activating t_{17} once may not be sufficient to find the fault. On the one hand, a large number of input parameter values could be used to invoke an operation for firing a transition, e.g., t_3 (Fig. 5). Each input leads to a distinct flight orientation, and only in a few specific orientations, the collision is likely to

⁶ When a collision is avoided, the copter is back to the flight mode. Hence, no testing interface needs to be invoked to trigger t_{12} . When the flight mode is changed back, a corresponding change event is generated by TM-Executor to activate the transition. As this event is from inside, we do not capture it in DFSM.

happen. On the other hand, the copter’s orientation is also affected by measurement uncertainties from sensors and actuation inaccuracy from actuators. Therefore, it requires a specific sequence of operation invocations and a specific sequence of uncertainty values to make the collision happen.

From numerous candidates, it is challenging to find the “right” operation invocations and sequence of uncertainty values to reveal a fault. Motivated by this, we present a fragility-oriented approach to find such cases to detect faults effectively.

4 Fragility-oriented testing under uncertainty

Invoking operations and introducing uncertainty are the two tasks that have to be fulfilled to detect faults in SH-CPS under uncertainty.

The task of operation invocation decides which operation to be invoked and which input parameter values to be used to drive the execution of the SUT. A sequence of operation invocations determines the path in the Dynamic Flat State Machine (DFSM) that the SUT will follow in executions. Besides, the model regulates the operations that can be invoked under each state.

The task of introducing uncertainty defines a concrete uncertainty-introduced environment, in which the SUT is tested. Whenever the SUT interacts with its environment via sensors or actuators, the environmental uncertainty may take effect, and thus, uncertainty values have to be generated and introduced into the SUT. For each uncertainty, its value can vary within a valid range. The combination of multiple uncertainties at different interaction points forms a great number of possible sequences of uncertainty values. A specific sequence of operation invocations has to work together with a specific sequence of uncertainty values to reveal a fault. To reduce the complexity of finding the two kinds of input, we adopt a two-step approach, as presented in Fig. 6.

The first step concentrates on finding a sequence of operation invocations that can make the SUT reach the most fragile state. During the first step, uncertainty values are only randomly generated. When an optimal sequence of invocations is found, it is used in the second step to drive the execution of SUT and test model, during which a sequence of uncertainty values will be found for fault revelation.

For the first step, we devise the fragility-oriented testing algorithm. By exploring various transitions in the test model and evaluating its consequent fragilities with multiple iterations,

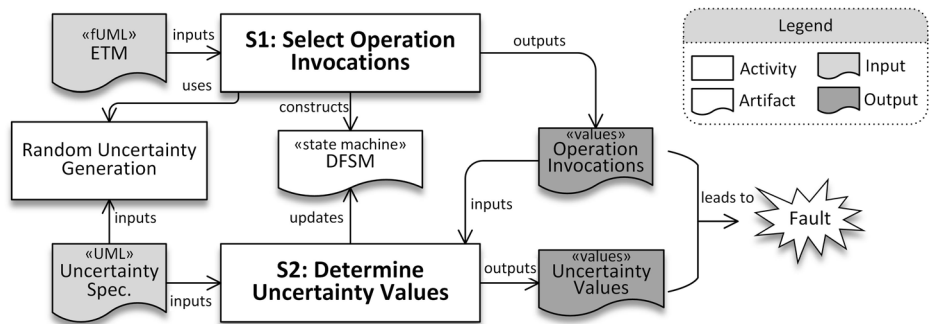


Fig. 6 Overview of fragility-oriented testing under uncertainty

the algorithm identifies the most fragile state and learns the shortest path to reach it. Accordingly, the sequence of operation invocations used to trigger the transitions in this path is selected as the optimal one and used in the next step.

In the second step, the sequence of invocations is fixed, and the uncertainty policy optimization algorithm gradually optimizes a parameterized uncertainty generation policy to find a corresponding sequence of uncertainty values that can reveal a fault. If a fault is detected, the transitions directly connected with the current active state are marked “faulty,” and it returns to the first step to find another invocation sequence without considering the “faulty” transitions. Otherwise, if no faults are detected in a certain number of executions, the fragilities corresponding to the states in the selected path are discounted by a discount factor. Based on the updated fragility, the fragility-oriented testing algorithm will recalculate the optimal sequence of invocations. Accordingly, the uncertainty policy optimization algorithm will try to find a corresponding sequence of uncertainty values again to detect faults.

The algorithms used in the two steps are presented in Section 5 and Section 6, respectively.

5 Fragility-oriented operation invocation

Definition 1 The fragility of the SUT in a given state s is a real value between 0 and 1, denoted as $F(s)$. It describes how close (distance wise) the state invariant of s is to be *false*, where 1 means that the state invariant is *false* and 0 means that it is far from being violated. We therefore define $F(s)$ as follows:

$$F(s) = 1 - \text{dis}(\neg\phi) \quad (1)$$

where $\neg\phi$ is the negation of state s 's invariant ϕ and $\text{dis}(\neg\phi)$ is a distance function (adopted from Ali et al. 2013) that returns a value between 0 and 1 indicating how close the constraint $\neg\phi$ is to be true. For instance, in the running example, if the SUT is currently in state *Avoiding2* and the value of state variable *threatDis* is 15, then the distance of invariant “*threatDis > 10*” to be false can be calculated as $\text{dis}(\neg(\text{threatDis} > 10)) = \frac{(15-10)+1}{(15-10)+1+1} = 0.86$, according to the distance function⁷ defined in Ali et al. (2013). The closer the distance is to zero, the higher the possibility the invariant is to be violated, i.e., the SUT failing in the state. Hence, $1 - \text{dis}(\neg\phi)$ is used to define the fragility of the SUT in state s .

Definition 2 The T value of a transition expressed as $T(t)$ is a real value between 0 and 1. It states the possibility that a fault can be revealed after firing the transition t . With an assumption that the more fragile the SUT is, the higher the chance a fault can be revealed, we define the T value of a transition as the discounted highest fragility of the SUT after firing the transition:

$$T(t) = \max_{s \in S_{\text{next}}} \{\gamma^n \cdot F(s)\} \quad (2)$$

where γ ($0 \leq \gamma < 1$) is a discount rate; S_{next} is a set of states that can be reached from t 's target state via a path in the DFSM, and n is the number of transitions between s and t 's target state. As for testing, revealing faults via a short path is preferable, we penalize the fragility of a state by multiplying γ^n , if traversing at least n transitions is required to reach the state from t 's target

⁷ The distance function of greater operator is $\text{dis}(x > y) = (y - x + k) / (y - x + k + 1)$, when $x \leq y$, where k is an arbitrary positive value. Here, we set $k = 1$. More details are in Ali et al. (2013).

state. For example, in Fig. 5, to obtain the T value of t_4 , we calculate the discounted fragility of the SUT in each state in S_{next} . For the fragility corresponding to *AvoidingI*, it needs to be discounted by γ^2 , since two transitions t_5 and t_9 have to be traversed to reach *AvoidingI* from t_4 's target state *Forward*. Clearly, the value of γ determines the importance of the state to be reached in the future.

5.1 Overview

The objective of the fragility-oriented testing (FOT) algorithm is to find the optimal operation invocation policy to find faults effectively. To achieve this objective, FOT tries to learn transitions' T values during the execution of the SUT. Each transition's T value indicates the possibility that a fault will be revealed after firing the transition. When transitions' T values are learned, by simply firing the transition with the highest T value, FOT can manage to find faults effectively. The pseudocode of FOT is presented below in Algorithm 1 (L1–L16).

In the beginning, all transitions' actual T values are unknown. As every transition has a possibility to reveal a fault, the estimated T value of each transition is initialized with the highest one (L1, L2). This encourages the algorithm to extensively explore uncovered transitions (Sutton and Barto 1998). After that, iterations of test execution and the learning process begin. At each iteration, the execution of the test model as well as the SUT starts from the initial state (L4) and terminates at a final state (L5). During the execution, a DFSM is dynamically constructed (L6) to enable the learning of T values. Whenever, the SUT enters a state s , FOT selects one of the outgoing transitions of s according to their estimated T values (L7, L8) and makes TM-Executor trigger the selected transition (L9). As the transition is fired, the system moves from s to s' . If the state invariant of s' is not satisfied, then a fault is detected (L11–L14). In this case, the current active state of the DFSM will be marked "faulty." Any transition connected with the faulty state will not participate in the transition selection and T value learning in the future.

Algorithm 1 *FOT(TMExecutor executor, ETM etm, int maxIteration):*

```

Input
  executor is TM-Executor, the testing framework
  etm is the Executable Test Model
  maxIteration is the maximum iteration number

Begin
  1 for each transition in etm
  2   transition.Tvalue  $\leftarrow$  1 // initialize T-values of transitions
  3 for i=1 to maxIteration
  4   etm.Start()
  5   while etm.ReachFinalState() is false
  6     dfsM  $\leftarrow$  EnrichDFSM(etm) // dynamically construct the DFSM
  7     reachedTransitions  $\leftarrow$  dfsM.activeState.outgoingTransitions
  8     selectedTransition  $\leftarrow$  SoftmaxSelect(reachedTransitions) //select transition
  9     executor.Trigger(selectedTransition)
 10    stateInvariant  $\leftarrow$  selectedTransition.target.invariant
 11    if executor.Evaluate(stateInvariant) is false
 12      LogFaultDetected(selectedTransition)
 13      dfsM.MarkFaultDetected(dfsM.activeState)
 14      break
 15    fragility  $\leftarrow$  1 - executor.DistanceToViolation(stateInvariant)
 16    executor.UpdateTvalue(selectedTransition, fragility) // revise the T-value of selectedTransition

End

```

If no invariant violation happens, the algorithm will evaluate the fragility of the SUT in s' (L15), i.e., $F(s')$, and use $F(s')$ to update estimated T values. Since it is possible to reach s' via numerous transitions, finding all these transitions and updating their T values are computationally impractical for a test model with hundreds of transitions. Thus, FOT only updates the estimated T value of the last triggered transition (L16). For instance, in the running example, when $t11$ is invoked and the state of the SUT changes to *Avoiding2*, FOT evaluates the value of $F(\text{Avoiding2})$ and uses the value to update the T value of $t11$, i.e., $T(t11)$. Since $F(s')$ is not a constant value, the upper bound of $F(s')$ is used to update the T value. As the iteration of the execution proceeds, the estimated T values are continuously updated and getting close to their actual values. In this way, the T values are learned from the execution and the learned T values direct FOT to effectively find faults. Note that testing budget determines the maximum number of iterations. If it is too small, FOT may not be able to find faults. The details of T value learning and transition selection policy are explained in the next two subsections, respectively.

5.2 T value learning

Before executing the SUT and the Executable Test Model (ETM), the T value $T(t)$ of every transition is unknown. We adopt a reinforcement learning approach to learn $T(t)$ from execution. A fundamental property of $T(t)$ is that it satisfies a recursive relation, which is called the Bellman equation (Sutton and Barto 1998), as shown in the formula below:

$$T(t) = \max \left\{ F(s_{tar}), \gamma \cdot \max_{t' \in T_{suc}} T(t') \right\} \tag{3}$$

where s_{tar} is the target state of transition t ; T_{suc} represents a set of direct successive transitions whose source state is s_{tar} . This equation reveals the relation between the T values of a transition and its direct successive transitions. It states that the T value of t must be equal to the greater of two values: the fragility of t 's target state ($F(s_{tar})$) and the maximum discounted T value of t 's direct successive transitions ($\gamma \cdot \max_{t' \in T_{suc}} T(t')$). Given a DFSM, $T(t)$ is the unique solution to satisfy Eq. (3). So, we try to update the estimate of each T value to make it get increasingly closer to satisfy Eq. (3). When Eq. (3) is satisfied by the estimated T values for all transitions, it implies that the true $T(t)$ is learned.

Inspired by Q-learning (Sutton and Barto 1998), a reinforcement learning method, FOT uses the estimated T value $ET(t)$ to approximate $T(t)$, i.e., the true T value. $ET(t)$ is updated in the following way to make it approach $T(t)$.

$$ET(t)' = \max \left\{ F(s_{tar}), \gamma \cdot \max_{t' \in T_{suc}} ET(t') \right\} \tag{4}$$

where $ET(t)'$ denotes the updated estimate of t 's T value and $ET(t_{suc})$ represents the current estimated T value of a successive transition.

Equation (4) enables FOT to iteratively update $ET(\mathfrak{t})$. Once a transition \mathfrak{t} is triggered, the fragility of the SUT in \mathfrak{t} 's target state $F(\mathfrak{s}_{\text{tar}})$ can be evaluated using Eq. (1). Using Eq. (4), $ET(\mathfrak{t})$ can be updated whenever a fragility value is obtained. As proved in Sutton and Barto (1998), as long as the estimated T values are continuously updated, $ET(\mathfrak{t})$ will converge to the true T value: $T(\mathfrak{t})$.

However, the fragility of the SUT in a state dynamically changes, due to the variation of test inputs and environmental uncertainty. To deal with this, we use the bootstrapping technique (Mooney et al. 1993) to predict the distribution of the fragility and select the upper bound of its 95% interval as the value for $F(\mathfrak{s}_{\text{tar}})$, to update the estimated T value. Thus $ET(\mathfrak{t})$ is iteratively updated by the following equation:

$$ET(\mathfrak{t})' = \max \left\{ \text{Upper}[F(\mathfrak{s}_{\text{tar}})], \gamma \cdot \max_{\mathfrak{t}_{\text{suc}} \in \mathbb{T}_{\text{suc}}} ET(\mathfrak{t}_{\text{suc}}) \right\} \quad (5)$$

where $\text{Upper}[F(\mathfrak{s}_{\text{tar}})]$ is the upper bound of $F(\mathfrak{s}_{\text{tar}})$'s 95% confidence interval.

5.3 Softmax transition selection

To effectively find faults, FOT should extensively explore different paths in a DFSM. Meanwhile, the covered high T value transitions should be exploited (triggered) more frequently to find faults, as a high T value implies a high possibility to reveal faults. Hence, in FOT, we use a softmax transition selection policy to address the dilemma of exploration and exploitation (Kaelbling et al. 1996) by assigning a selection probability to a transition proportional to the transition's T value. The selection probability is given below (from Sutton and Barto 1998):

$$\text{Prob}(\mathfrak{t}'_{\text{out}}) = e^{ET(\mathfrak{t}'_{\text{out}})/\tau} / \sum_{\mathfrak{t}_{\text{out}} \in \mathbb{T}_{\text{out}}} e^{ET(\mathfrak{t}_{\text{out}})/\tau} \quad (6)$$

where $\text{Prob}(\mathfrak{t}'_{\text{out}})$ denotes the selection probability of an outgoing transition $\mathfrak{t}'_{\text{out}}$; $ET(\mathfrak{t}'_{\text{out}})$ is the estimated T value; \mathbb{T}_{out} represents the set of all outgoing transitions under the current DFSM state, and τ is a parameter, called temperature (Anzai 2012). τ is a positive real value from 0 to infinity. A large τ causes transitions to be equally selected, whereas a small τ causes high T value transitions to be selected much more frequently than transitions with lower T values.

In the beginning, all transitions' estimated T values ($ET(\mathfrak{t})$) are initialized to 1; thus, initially, transitions have equal probability to be selected. As testing proceeds, $ET(\mathfrak{t})$ is continuously updated using Eq. (5). Directed by $ET(\mathfrak{t})$, the softmax policy assigns a high selection probability to transitions that lead to states with high fragilities. As a result, more fragile states will be exercised more frequently. Note that this does not preclude covering the less fragile states. In addition, loops in the test model are also covered depending on fragilities of states involved in a loop.

6 Uncertainty policy optimization

When a sequence of operation invocations is selected, the sequence determines the path in the DFSM that the SUT will follow in test executions. Besides the operation invocations, a sequence of uncertainty values is required to execute the SUT under uncertainty. Due to the effect of uncertainty and the execution of the SUT, the state variables of the SUT constantly vary within a range in each state. Consequently, every state s_i in the execution path corresponds to a number of state instances $\{s_{i1}, s_{i2}, \dots, s_{ik}\}$.

Definition 3 A state instance, s_{ij} , is an instance of an abstract state, s_i , in a DFSM. The state instance reflects the SUT’s actual state at a specific time point, and the state instance is represented by the values of the SUT’s all state variables, i.e., $s_{ij} = \{v_1, v_2, \dots, v_{nsv}\}$. Based on the definition of fragility, we define the fragility of the SUT in a given state instance as follows:

$$F(s_{ij}) = F(s_i) = 1 - dis(\neg \circ_i) \tag{7}$$

Note that although the state invariant \circ_i of a state s_i constrains only a few state variables, the other state variables may have impacts on the constrained variables. Therefore, all state variables may have direct or indirect effects on the fragility. Due to this reason, we employ the values of all state variables to represent the state instance s_{ij} .

Figure 7 illustrates the relationship between state and state instance. The number of state instances corresponding to a state depends on the number of environmental interactions that the SUT performs in the state. For instance, after the operation *arm* is invoked, ArduCopter takes 1 s to enter the next state *Started*. Within 1 s, the copter reads sensor and controls actuators 400 times. Consequently, the state *Stopped* corresponds to 400 state instances.

As the behavior of the SUT has been determined by the selected operation invocations, the uncertainty values u_{ij} decide the next state instance, s_{ij+1} , and the SUT will switch to from the previous instance, s_{ij} . To effectively detect faults, the optimal uncertainty values should be used to maximize the fragility of the SUT. To effectively find the optimal uncertainty values, this section presents the UPO algorithm.

6.1 Uncertainty generation policy

To effectively explore various uncertainty values, we propose to use a parameterized policy $\pi_\theta(u_{ij}|s_{ij})$ to decide the uncertainty values. The policy $\pi_\theta(u_{ij}|s_{ij})$ determines

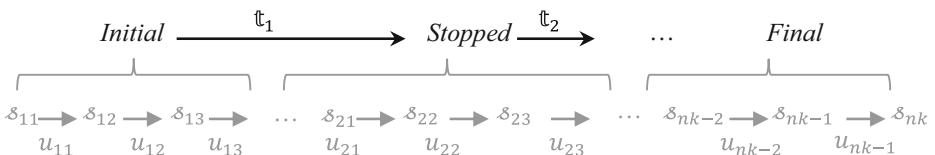


Fig. 7 Relations between states and state instances

the probability distribution of u_{ij} given the condition that \mathcal{s}_{ij} is the current state instance. The conditional probability distribution can be changed, by adjusting policy parameters θ .

As artificial neural network (ANN) has been demonstrated to be an effective decision-making mechanism (Aronson et al. 2005), we adopt ANN as the parameterized policy for uncertainty generation.

An ANN, used for uncertainty generation, takes a state instance as input. Each neuron in the input layer represents a state variable of the SUT. The input neurons make the values of state variables traverse the ANN. Based on the received values, the output neurons calculate the final output. Each output determines the probability distribution of one uncertainty, under the condition that the current state instance is the one fed to the input layer. Inspired by an existing algorithm (Schulman et al. 2015), we use a truncated normal distribution as the conditional probability distribution. The mean value of the distribution is the output value, and the value of its variance is a constant positive value ε . By sampling from the distribution, we can decide the uncertainty values to be used for each state instance. Figure 8 presents an example of the ANN used for the running example. The ANN receives the values of all state variables. The values are processed by the interconnected neurons and are mapped to a truncated normal distribution for each uncertainty.

To effectively find faults, we need to optimize the parameterized policy so that the uncertainty values generated from the policy can increase the fragility of the SUT and make the system likely to fail. The parameters of the policy include the weights of connections and bias values of neurons. Except them, the numbers of layers and the neurons in hidden layers, the activation function, and the output function are all predefined. Once being set up, they are fixed. The next section explains an iterative approach to optimize the policy concerning these parameters.

6.2 Policy optimization

The goal of policy optimization is to tune the parameters of the uncertainty generation policy, to maximize the fragility of the SUT. The policy $\pi_{\theta}(u_{ij}|\mathcal{s}_{ij})$ determines the uncertainty values to be introduced in each state instance. When the state instance \mathcal{s}_{ij}

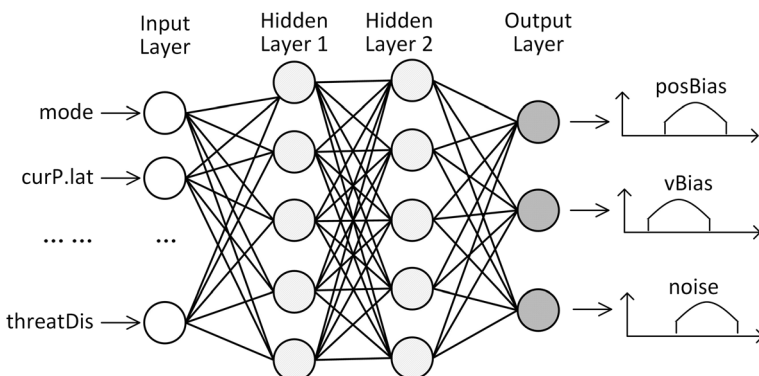


Fig. 8 Example of uncertainty generation policy

and uncertainty values u_{ij} are given, the next state instance \mathcal{s}_{ij+1} is determined, as shown in Fig. 7. Since the execution of the SUT always starts from the same initial state instance, when the sequence of operation invocations is fixed, the uncertainty generation policy $\pi_\theta(u_{ij}|\mathcal{s}_{ij})$ also determines the probability distribution of the state instance. It means the policy decides the probability that a state instance \mathcal{s}_{ij} can be reached by the SUT in an execution. Based on this, we formally express the goal function as follow (Schulman et al. 2015):

$$\eta(\pi_\theta) = \mathbb{E}_{\mathcal{s}_{ij} \sim \rho_\theta, u_{ij} \sim \pi_\theta} [T_\theta(\mathcal{s}_{ij}, u_{ij})] \tag{8}$$

where \mathbb{E} denotes the expectation of the highest discounted fragility, $T_\theta(\mathcal{s}_{ij}, u_{ij})$, that can be obtained by following a given policy π_θ . ρ_θ denotes the probability distribution of the state instance, which is controlled by the parameters of the policy. $T_\theta(\mathcal{s}_{ij}, u_{ij})$ denotes the highest discounted fragility that can be reached after introducing uncertainty values u_{ij} in a state instance \mathcal{s}_{ij} :

$$T_\theta(\mathcal{s}_{ij}, u_{ij}) = \max_{k \in [1, +\infty)} \gamma^k \cdot F(\mathcal{s}_{ij+k}) \tag{9}$$

Since the value of $T_\theta(\mathcal{s}_{ij}, u_{ij})$ depends on the state instances that are to be covered after \mathcal{s}_{ij} and the policy determines the following states, the value of $T_\theta(\mathcal{s}_{ij}, u_{ij})$ also relies on the policy. As a result, both ρ_θ and $T_\theta(\mathcal{s}_{ij}, u_{ij})$ depend on the parameters of the policy.

Suppose we have two policies: π_θ and $\pi_{\theta'}$. To compare them, we have to apply both policies to execute the SUT a number of times, and derive the distributions of state instance and the values of highest discounted fragility from the execution. Since the cost to execute the SUT is relatively high, it is difficult to find the direction for improvement directly based on Eq. (8).

To simplify the optimization problem, we choose to optimize an approximation of the goal function (Kakade and Langford 2002):

$$\eta(\pi_\theta) \approx L(\pi_\theta) = \mathbb{E}_{\mathcal{s}_{ij} \sim \rho_{\theta'}, u_{ij} \sim \pi_\theta} [T_{\theta'}(\mathcal{s}_{ij}, u_{ij})] \tag{10}$$

Note that ρ_θ is changed to $\rho_{\theta'}$ and $T_\theta(\mathcal{s}_{ij}, u_{ij})$ is changed to $T_{\theta'}(\mathcal{s}_{ij}, u_{ij})$. This allows us to directly find the optimal improvement direction based on the $\rho_{\theta'}$ and $T_{\theta'}(\mathcal{s}_{ij}, u_{ij})$ obtained from an existing policy $\pi_{\theta'}$, without extra executions. The general idea is that the value of $T_{\theta'}(\mathcal{s}_{ij}, u_{ij})$ points out the expected reward of introducing uncertainty values u_{ij} in a given state \mathcal{s}_{ij} . To increase the total expectation, we just need to adjust the parameters of the policy to increase the probability to generate u_{ij} in \mathcal{s}_{ij} , if $T_{\theta'}(\mathcal{s}_{ij}, u_{ij})$ is high. As proven in Schulman et al. (2015), as long as the Kullback–Leibler divergence, a distance measure, between the two policy π_θ and $\pi_{\theta'}$ is bounded by a constant step size, the true reward function $\eta(\pi_\theta)$ is guaranteed to be improved.

To further simplify the calculation of Eq. (10), we replace the expectation over the uncertainty values following π_θ by the expectation over the uncertainty values following $\pi_{\theta'}$, according to importance sampling (Glynn and Iglehart 1989):

$$L(\pi_\theta) = \mathbb{E}_{\mathcal{s}_{ij} \sim \rho_{\theta'}, u_{ij} \sim \pi_{\theta'}} \left[\frac{\pi_\theta(u_{ij}|\mathcal{s}_{ij})}{\pi_{\theta'}(u_{ij}|\mathcal{s}_{ij})} \cdot T_{\theta'}(\mathcal{s}_{ij}, u_{ij}) \right] \tag{11}$$

Based on this, we propose the following policy optimization algorithm, as given in Algorithm 2. The general idea is that whenever we find a sequence of uncertainty values $u_{11}, u_{12}, \dots, u_{nm}$ that leads to a high fragility, i.e., their $T_{\theta}(s_{ij}, u_{ij})$ is high, we adjust θ' to θ to increase the probability to generate the uncertainty sequence.

Initially, an ANN, used as the uncertainty generation policy, is constructed (L1). After initializing the vectors used for saving samples of states, uncertainty values, and discounted fragilities, the iteration of execution begins (L7). During execution, uncertainty values are generated by the policy (L10) and are introduced to the SUT (L11) to run it under uncertainty. Affected by the uncertainty values, the state of the SUT switches to another one. The fragility of the SUT in the new state is evaluated and discounted by a discount factor (L13). If the discounted fragility exceeds the highest one that has been found so far, it means that a better sequence of uncertainty values is found to make the SUT more fragile (L17). In this case, we apply the conjugate gradient algorithm (Hestenes and Stiefel 1952) to adjust the parameter values of the policy to maximize the generation probability of the sequence of uncertainty values (L20). After that, the updated policy is used in the following execution to find an even better sequence of uncertainty values.

Algorithm 2 *UPO(TMExecutor executor, UnGenerator generator, int numStateVar, int numUncer, int maxIter):*

Input *executor* is TM-Executor, the testing framework
generator is the uncertainty generator
numStateVar is the number of state variables
numUncer is the number of uncertainties
maxIteration is the maximum iteration number

Begin

```

1 policy.Init(numStateVar, numUncer)
2 highestDisFragility = 0
3 for i=1 to maxIteration
4   states ← []
5   uncertainties ← []
6   discountedFragilities ← []
7   executor.StartExecution()
8   while not executor.Finish()
9     s ← executor.CurrentSUTState()
10    u ← policy.Sample(s)
11    generator.IntroduceUncertainties(u)
12    s' ← executor.CurrentSUTState()
13    f ← executor.ComputeDiscountedFragility(s')
14    states.Append(s)
15    uncertainties.Append(u)
16    discountedFragilities.Append(f)
17    if f > highestDisFragility
18      highestDisFragility = f
19  if highestDisFragility is changed
20    policy.Update(states, uncertainties, discountedFragilities)

```

End

7 Implementation

We implemented the fragility-oriented testing approach in TM-Executor. Figure 9 presents its three packages: software in the loop testing (light gray), uncertainty generation (dark gray), and FOT (white).

TM-Executor tests the software of the SUT in a simulated environment. During testing, sensor data is computed by simulation models in simulators. Based on the simulated data, the software generates actuation instructions to control the system. Uncertainties are added to simulators’ input and output to simulate the effects of uncertainties. Based on the valid range of each uncertainty, the UPO algorithm generates uncertainty values whenever sensor data or actuation instructions are transferred between the software and simulators. By using the values to modify simulators’ inputs and outputs, the uncertainties are introduced into the testing environment.

The SUT and its Executable Test Model (ETM) are executed together by an execution engine, which is deployed in Moka (Tatibouet 2016), a UML model execution platform. During the execution, the engine dynamically derives a DFSM from the test model and uses it to guide the execution. Meanwhile, the active state’s state invariant is checked by a test inspector (using Dresden OCL (Demuth and Wilke 2009)). The inspector evaluates the invariant with the actual values of the state variables, which are updated by the execution engine via testing interfaces (Section 2.3). If the invariant is evaluated to be false, a fault is detected. Otherwise, the inspector calculates the fragility of the SUT in the current state, using Eq. (1). Taking fragility as input, the FOT algorithm updates its estimate of T value (Eq. (5)) and uses the softmax policy to select the next transition. Next, the test driver generates a valid test input with EsOCL (Ali et al. 2013), a search-based test data generator, for firing the selected transition. The execution engine takes this input to invoke the corresponding operation, causing the ETM and the SUT to enter the next state. In this way, T values are learned from iterations of execution and the learned T values direct FOT to effectively find faults.

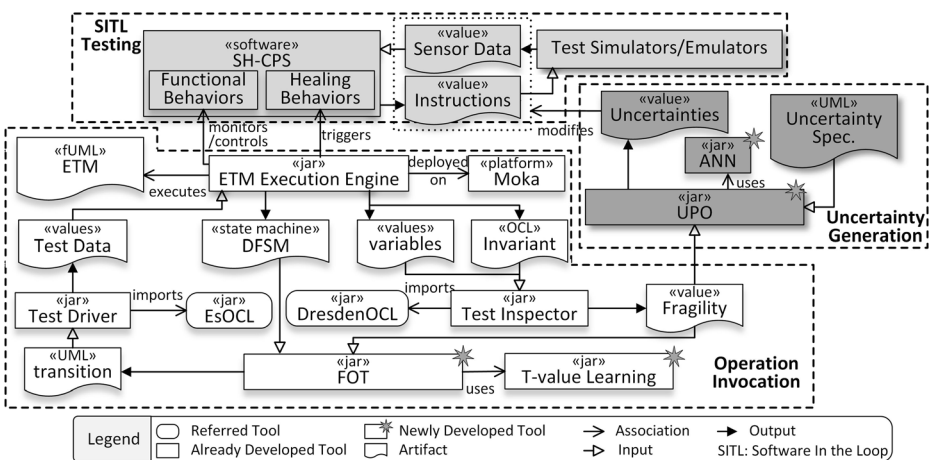


Fig. 9 SH-CPS testing framework (Ma et al. 2018)

8 Evaluation

This section presents the performance evaluation of FOT and UPO, including experiment design in Section 8.1, experiment execution in Section 8.2, experiment results in Section 8.3, the discussion in Section 8.4, and threats to validity in Section 8.5.

8.1 Experiment design

This section presents the design of the experiment, by following three carefully defined research questions.

8.1.1 Research questions

RQ1: Does FOT+UPO have the highest fault detection ability for testing SH-CPSs under uncertainty?

Since testing SH-CPSs under uncertainties comprises two tasks, i.e., invoking operations and introducing uncertainties, we devise FOT and UPO to address them, respectively. To assess their performance, we select a baseline algorithm for each of them. For FOT, we choose a coverage-oriented testing (COT) algorithm as the baseline since it is a prevalent approach applied in the testing of CPSs (Asadollah et al. 2015). This algorithm selects operation invocations based on the coverage frequencies of transitions, and its aim is to evenly traverse each transition. With respect to uncertainty generation, UPO is compared with a random approach. In this approach, uncertainty values are just generated from probability or possibility distributions. As a result, we have two algorithms, i.e., FOT and COT, for selecting operation invocations, and two algorithms, i.e., UPO and Random (R), for generating uncertainty values. In total, we obtain four approaches: FOT+UPO, FOT+R, COT+UPO, COT+R. We apply them to test three SH-CPSs to check which approach can detect more faults in SH-CPSs under uncertainties.

RQ2: To what extent the fault detection ability can be enhanced by FOT and UPO, compared with the others?

With this research question, we aim to investigate the effectiveness of FOT + UPO, i.e., assess the percentage of improvement regarding fault detection ability achieved by FOT and UPO compared with the other three approaches.

RQ3: Concerning the optimal testing approach, what are the correlations between fault detection ability, testing time, and the scale of uncertainty variation?

The number of detected faults not only depends on the fault detection ability of a testing approach but also relies on the variation range of each uncertainty and the amount of time that can be used by the testing approach. This research question helps us reveal whether more faults can be detected as the testing time and the scale of uncertainty variation increase.

Table 1 Identified uncertainties from the three case studies

| Hardware | Uncertainty | Range | Hardware | Uncertainty | Range |
|---------------|--------------------|--|-----------|-------------------|--------------------------|
| Accelerometer | Noise | (− 9 mg, + 9 mg) | GPS | Position accuracy | (− 2.5 m, + 2.5 m) |
| | Non-linearity | (− 0.5%, + 0.5%) | | Velocity accuracy | (− 0.05 m/s, + 0.05 m/s) |
| Motor | Rotation noise | (− 0.3°, + 0.3°) | Gyroscope | Noise | (− 0.3°/s, + 0.3°/s) |
| | Acceleration noise | (− 0.02 m/s ² , + 0.02 m/s ²) | Barometer | Accuracy | (− 150 Pa, + 150 Pa) |

8.1.2 Case studies

We used three open-source SH-CPSs for evaluation: (1) ArduCopter is a fully featured copter control system supporting 18 flight modes to control a copter. It has five self-healing behaviors to avoid crash and collision; (2) ArduRover⁸ is an autopilot system for ground vehicles having two self-healing behaviors to avoid an obstacle and handle the disruption of control link; (3) ArduPlane⁹ is an autonomous plane control system having two self-healing behaviors to avoid collision and address network disruption. Test execution was performed with software in the loop simulators, including GPS, barometer, accelerometer, gyroscope, and motor simulators. Nine fault injection operations were implemented in the simulators to trigger the nine self-healing behaviors to test them in the presence of uncertainty.

The three SH-CPSs are affected by eight uncertainties related to the sensors and actuators. Based on the product specifications of the sensors and actuators, we specified their variation range, as presented in Table 1.

Before testing, we built the Executable Test Model (ETM) for each self-healing behavior of the three case studies. Table 2 shows the statistics of the ETMs. Moreover, we examined the average amount of time that the testing framework takes to execute the ETM and the SUT from their initial state to a final state, as presented in the last row of Table 2. Note that the average execution times are determined by the implementation of the SUTs, and they are not affected by different testing approaches.

8.1.3 Experiment tasks

Three tasks have to be performed to address the three research questions. Table 3 gives an overview of the three tasks.

For RQ1, T_1 is performed to investigate which testing approach can detect more faults in the nine self-healing behaviors under the eight uncertainties. To reduce the impact of testing time and the scale of uncertainty variation, we choose three settings for each of them. The three testing times are 72, 144, and 216 h. This allows each testing approach to execute the ETM and the SUT approximately 500, 1000, and 1500 times to find faults. Meanwhile, we choose three scales of uncertainty variation: 80%, 100%, and 120%. One hundred percent represents the standard

⁸ <http://ardupilot.org/rover/>

⁹ <http://ardupilot.org/plane/>

Table 2 Descriptive statistics of ETMs

| | ArduCopter | | | | | ArduRover | | ArduPlane | |
|----------------------|------------|------|------|------|------|-----------|------|-----------|------|
| | ETM1 | ETM2 | ETM3 | ETM4 | ETM5 | ETM6 | ETM7 | ETM8 | ETM9 |
| #States | 64 | 60 | 70 | 64 | 36 | 58 | 54 | 79 | 40 |
| #Transitions | 440 | 268 | 286 | 440 | 106 | 306 | 303 | 347 | 104 |
| Avg. Exe. time (min) | 8 | 9 | 7 | 8 | 8 | 10 | 11 | 6 | 6 |

variation ranges shown in Table 1. Eighty percent (120%) means reducing (increasing) the ranges by 20%. For instance, the 80%, 100%, and 120% variation ranges of the uncertainty *Noise* from the accelerometer are $(-7.2 \text{ mg}, +7.2 \text{ mg})$, $(-9 \text{ mg}, +9 \text{ mg})$, and $(-10.8 \text{ mg}, +10.8 \text{ mg})$, respectively. The ranges are only modified by 20% to avoid making the uncertainty variation ranges deviate too far from the reality. The differences in uncertainty scale help us reveal the fault detection ability of a testing approach under different uncertainty scales.

As a result, the nine self-healing behaviors are tested with the four approaches in nine different test settings. In total, 81 testing jobs are to be conducted for each of the four approaches. Moreover, each testing job is performed ten times to reduce the impact of randomness.

Regarding RQ2, T_2 is performed to calculate the percentage of improvement in terms of fault detection when the optimal testing approach is applied.

For RQ3, we conduct T_3 to analyze the correlations among fault detection ability, testing time, and the scale of uncertainty variation.

Table 3 Overview of experiment design

| RQ | Task | Metric | Statistical test | Testing approach | Testing time (h) | Scale of uncertainty variation | Case studies |
|----|---|------------|---|--|---------------------------|--------------------------------|---|
| 1 | T_1 : compare the fault detection ability of the selected testing approaches | <i>NDF</i> | Mann–Whitney <i>U</i> test, Vargha and Delaney's \hat{A}_{12} | 1: FOT+UPO 2: FOT+R 3: COT+UPO 4: COT+R | 1: 72 2: 144 3: 216 | 1: 80% 2: 100% 3: 120% | 1: ArduCopter (5 SH behaviors) 2: ArduPlane (2 SH behaviors) |
| 2 | T_2 : calculate the improvement with respect to the percentage of faults detected by the optimal approach | <i>PDF</i> | N/A | | | | 3: ArduRover (2 SH behaviors) |
| 3 | T_3 : Analyze the correlations between the fault detection ability, testing time, and uncertainty scale | | N/A | The optimal one | | | |

8.1.4 Evaluation metrics and statistics tests

RQ1: We define the *number of detected faults (NDF)* to quantify the fault detection ability of each testing approach. *NDF* is the number of faults that are detected by an approach in a self-healing behavior within limited testing time. As the first step of analyzing the results, we applied the Shapiro–Wilk test with a significance level of 0.05 to check the normality of *NDF* values. Results show that the distribution of the *NDF* values departs from normality. Therefore, we use the non-parametric Mann–Whitney *U* test with the significant level of 0.05 to determine the significance of differences between two testing approaches. That is, a comparison result is statistically significant if the *p value* is less than 0.05. Furthermore, following the guideline in Arcuri and Briand (2011), we apply Vargha and Delaney’s \hat{A}_{12} statistics to measure the effect size, i.e., measure the probability that a testing approach *A* can detect more faults than another approach *B*. If *A* and *B* are equivalent, \hat{A}_{12} equals 0.5. If \hat{A}_{12} is greater than 0.5, then *A* has higher chance to detect more faults than *B*.

RQ2: To calculate to what extent the fault detection ability can be enhanced by the optimal testing approach, compared with the others, we define another metric: the *percentage of detected faults (PDF)*, that is, the percentage of faults in one self-healing behavior that can be detected by a testing approach. It is calculated as follows:

$$PDF_i = \frac{NDF_i}{Total_i}$$

where NDF_i is the number of faults detected in one testing job for the *i*th self-healing behavior, and $Total_i$ is the total number of faults detected in the behavior. Because the number of actual faults cannot be determined, the total number of detected faults is used instead. This metric normalizes the number of detected faults, which enables us to compare the performance of each approach across different self-healing behaviors.

RQ3: We also use *PDF* as a measure of fault detection ability to analyze its relations with testing time and the scale of uncertainty variation. Here, we are interested in assessing the monotonic relations among them, i.e., whether more faults can be detected, as the testing time and the scale of uncertainty increase. Consequently, we apply box plot to present the distribution of *PDF* under each testing time and uncertainty scales, based on five numbers: minimum, first quartile, median, third quartile, and maximum. In the plot, a rectangle spans the first quartile to the third quartile. A mark inside the rectangle indicates the median, and the lines above and below the rectangle denote the maximum and minimum.

8.2 Experiment execution

We implemented the proposed algorithms in the TM-Executor (Ma et al. 2018). As explained in Section 4, each testing approach consists of two steps: selecting a sequence of operation invocations and finding a sequence of uncertainty values. The number of iterations for the first step is 200, and the maximum iteration number for the second step is 500. For FOT, we set discount rate γ to 0.99 and temperature τ to 0.2. For UPO, the ANN used as uncertainty generation policy contains three layers: an input layer, an output layer, and a hidden layer. The number of input neurons is equal to the number of

state variables of each SUT, and the number of hidden neurons is two times the number of input neurons. The number of output neurons is equal to the number of uncertainties. The variance ε used by the stochastic policy is 0.4. These are commonly used settings in reinforcement learning (Duan et al. 2016).

The experiment is executed on Abel, a computer cluster at the University of Oslo.¹⁰ Each testing job is run with eight cores and 32 GB RAM.

8.3 Experiment results

RQ1: Table 4 presents the average number of faults detected by each testing approach (task T₁). From the table, we can observe that FOT+UPO detected more faults than the other three approaches for all the case studies and test settings. We further conducted a statistical test to determine whether such results are statistically significant.

Table 5 summarizes the results of comparing the *NDF* achieved by FOT+UPO against those achieved by FOT+R, COT+UPO, and COT+R. FOT+UPO significantly outperformed the other approaches in 73 out of 81 testing jobs, as the values of \hat{A}_{12} are greater than 0.5 and 73 *p* values are less than 0.05. For the other eight cases, since the scale of uncertainty is low, the four testing approaches only detected few faults within 72 h. Thus, there is no significant difference among the four approaches.

Therefore, the answer to RQ1 is that among the four testing approaches, FOT+UPO has the highest fault detection ability for testing SH-CPSs under uncertainties. Compared with the others, FOT+UPO detected significantly more faults in 73 out of 81 testing jobs.

RQ2: To compare the fault detection ability of each testing approach across the nine self-healing behaviors, we calculated the *PDF* by dividing the *NDF* by the total number of faults detected in the experiment (task T₂). Table 6 presents the results. In most cases, FOT+UPO detected more than 70% of faults, while FOT+R and COT+UPO merely detected less than 17% of faults. For COT+R, it only detected one fault once in a self-healing behavior of ArduRover.

Therefore, we answer RQ2 as compared with COT and random uncertainty generation, FOT and UPO together can enhance the fault detection ability of a testing approach by at least 50%. FOT+UPO detected over 70% faults in most cases, whereas FOT+R, COT+UPO, and COT+R at most detected 17%, 3%, and 1% faults on average, respectively.

RQ3: Using box plot, we investigated the correlations among the fault detection ability (*PDF*) of FOT+UPO, testing time (*TT*), and the scale of uncertainty variation (*SU*). As shown in Fig. 10, when *SU* is 0.8 or 1.0, *PDF* tends to increase as *TT* grows from 72 to 216 h. The tendency becomes less significant when *SU* is increased to 1.2. This indicates that when *SU* is relatively low, the testing approach needs to take 216 h to detect all the faults, whereas when *SU* is high, 144 h are sufficient for the testing approach to find most faults.

Regarding the correlation between *PDF* and *SU*, it exposes similar phenomena. When *TT* is 72 h, the median of *PDF* increases from 0 to 0.2, as *SU* grows from 0.8 to 1.2. When *TT* is extended to 144 h, their positive relation becomes less significant. As the testing approach has sufficient time to detect most faults, there is no significant difference in *PDF* for different *SUs*.

Therefore, we answer RQ3 as *PDF* is positively correlated with both *TT* and *SU*. As *TT* or *SU* increases, *PDF* tends to increase as well. However, when *SU* is high, or *TT* is long, the

¹⁰ <http://www.uio.no/english/services/it/research/hpc/abel/>

Table 4 Average number of faults detected by each approach

| Setting | Approach | ArduCopter | | | | | ArduPlane | | ArduRover | | Avg. |
|---------|----------|------------|-----|-----|-----|-----|-----------|-----|-----------|-----|------|
| | | SH1 | SH2 | SH3 | SH4 | SH5 | SH1 | SH2 | SH1 | SH2 | |
| S1 | FOT+UPO | 0.3 | 0.4 | 0.3 | 0.3 | 0.2 | 0.7 | 0.4 | 0.3 | 0.2 | 0.3 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2 | FOT+UPO | 1.3 | 1.8 | 1.5 | 1.7 | 1.8 | 1.9 | 1 | 1 | 1 | 1.4 |
| | FOT+R | 0.1 | 0 | 0 | 0 | 0 | 0 | 0 | 0.1 | 0.2 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S3 | FOT+UPO | 2.1 | 2.4 | 2 | 2.2 | 2.1 | 1.9 | 1 | 1 | 1 | 1.7 |
| | FOT+R | 0.1 | 0 | 0 | 0 | 0 | 0.1 | 0.1 | 0.2 | 0.4 | 0.1 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S4 | FOT+UPO | 0.5 | 0.6 | 0.4 | 0.5 | 0.5 | 1.1 | 0.8 | 0.9 | 0.9 | 0.7 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S5 | FOT+UPO | 1.2 | 1.9 | 1.9 | 2.1 | 2 | 2 | 1 | 1 | 1 | 1.6 |
| | FOT+R | 0.1 | 0 | 0 | 0.1 | 0 | 0.2 | 0.1 | 0.1 | 0.3 | 0.1 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S6 | FOT+UPO | 2.3 | 2.6 | 2.5 | 2.3 | 2.4 | 2 | 1 | 1 | 1 | 1.9 |
| | FOT+R | 0.3 | 0 | 0.1 | 0.1 | 0 | 0.3 | 0.1 | 0.3 | 0.5 | 0.2 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.1 | 0.1 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S7 | FOT+UPO | 0.9 | 0.6 | 0.5 | 0.5 | 0.6 | 1.3 | 0.9 | 1 | 1 | 0.8 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S8 | FOT+UPO | 2 | 2.5 | 2.3 | 2.5 | 2.7 | 2 | 1 | 1 | 1 | 1.9 |
| | FOT+R | 0.1 | 0 | 0.1 | 0 | 0.1 | 0.3 | 0.1 | 0.2 | 0.3 | 0.1 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S9 | FOT+UPO | 2.8 | 2.9 | 3.1 | 2.8 | 3 | 2 | 1 | 1 | 1 | 2.1 |
| | FOT+R | 0.3 | 0.1 | 0.2 | 0 | 0.1 | 0.4 | 0.2 | 0.4 | 0.5 | 0.2 |
| | COT+UPO | 0 | 0.1 | 0 | 0 | 0 | 0.1 | 0 | 0.1 | 0.1 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.1 | 0 | 0 |

S1, test 72 h with 80% uncertainty range; S2, test 144 h with 80% uncertainty range; S3, test 216 h with 80% uncertainty range; S4, test 72 h with 100% uncertainty range; S5, test 144 h with 100% uncertainty range; S6, test 216 h with 100% uncertainty range; S7, test 72 h with 120% uncertainty range; S8, test 144 h with 120% uncertainty range; S9, test 216 h with 120% uncertainty range

positive correlations become less significant. Since *PDF* reaches a relatively high value earlier, it cannot be further significantly promoted.

8.4 Discussion

Based on the results of the experiment, we have three key observations. First, due to the effect of uncertainties, self-healing behaviors might fail to timely detect faults or improperly adapt system behaviors. For instance, because of sensors' measurement uncertainties, the copter could not accurately capture its location, orientation, and velocity. When the copter was about to collide with another vehicle, inaccurate measurements sometimes caused the copter to

Table 5 Results of comparing the approaches for testing each self-healing behavior

| Setting | Compared with | | | ArduCopter | | | | | | ArduPlane | | | | | | ArduRover | | | | | | | | |
|---------|----------------|----------|----------------|------------|----------------|----------|----------------|----------|----------------|-----------|----------------|----------|----------------|----------|----------------|-----------|----------------|------|----------------|----------|----------------|----------|-------|----------|
| | SH1 | | SH2 | | SH3 | | SH4 | | SH5 | | SH1 | | SH2 | | SH1 | | SH2 | | SH1 | | SH2 | | | |
| | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | \hat{A}_{12} | P | | |
| S1 | FOT+R | 0.6 | 0.37 | 0.7 | 0.07 | 0.65 | 0.15 | 0.65 | 0.15 | 0.6 | 0.35 | 0.85 | 0.01 | 0.7 | 0.07 | 0.65 | 0.15 | 0.6 | 0.35 | 0.65 | 0.15 | 0.6 | 0.35 | |
| | COT+UPO | 0.6 | 0.37 | 0.7 | 0.07 | 0.65 | 0.15 | 0.65 | 0.15 | 0.6 | 0.35 | 0.85 | 0.01 | 0.7 | 0.07 | 0.65 | 0.15 | 0.6 | 0.35 | 0.65 | 0.15 | 0.6 | 0.35 | |
| | COT+R | 0.6 | 0.37 | 0.7 | 0.07 | 0.65 | 0.15 | 0.65 | 0.15 | 0.6 | 0.35 | 0.85 | 0.01 | 0.7 | 0.07 | 0.65 | 0.15 | 0.6 | 0.35 | 0.65 | 0.15 | 0.6 | 0.35 | |
| S2 | FOT+R | 0.97 | 0.006 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.002 | 0.95 | 0.003 | 0.9 | 0.006 | 0.97 | 0.006 | <i>I</i> | 0.002 | 0.97 |
| | COT+UPO | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.002 | 0.95 | 0.003 | 0.9 | 0.006 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.002 | 0.95 | 0.003 | 0.9 | 0.006 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| S3 | FOT+R | 0.99 | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.003 | 0.99 | 0.004 | 0.95 | 0.003 | 0.9 | 0.006 | 0.8 | 0.019 | 0.99 | 0.005 | <i>I</i> | 0.002 | 0.99 |
| | COT+UPO | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.003 | <i>I</i> | 0.003 | <i>I</i> | 0.002 | 0.95 | 0.003 | 0.8 | 0.019 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.003 | <i>I</i> | 0.003 | <i>I</i> | 0.003 | <i>I</i> | 0.002 | 0.95 | 0.003 | 0.8 | 0.019 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| S4 | FOT+R | 0.75 | 0.037 | 0.8 | 0.019 | 0.7 | 0.072 | 0.75 | 0.037 | 0.75 | 0.037 | 0.75 | 0.037 | 0.9 | 0.006 | 0.95 | 0.003 | 0.95 | 0.003 | 0.75 | 0.037 | 0.8 | 0.019 | 0.75 |
| | COT+UPO | 0.75 | 0.037 | 0.8 | 0.019 | 0.7 | 0.072 | 0.75 | 0.037 | 0.75 | 0.037 | 0.75 | 0.037 | 0.9 | 0.006 | 0.95 | 0.003 | 0.95 | 0.003 | 0.75 | 0.037 | 0.8 | 0.019 | 0.75 |
| | COT+R | 0.75 | 0.037 | 0.8 | 0.019 | 0.7 | 0.072 | 0.75 | 0.037 | 0.75 | 0.037 | 0.75 | 0.037 | 0.9 | 0.006 | 0.95 | 0.003 | 0.95 | 0.003 | 0.75 | 0.037 | 0.8 | 0.019 | 0.75 |
| S5 | FOT+R | 0.96 | 0.006 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | 0.99 | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | 0.96 | 0.006 | <i>I</i> | 0.002 | 0.96 |
| | COT+UPO | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| S6 | FOT+R | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.85 | 0.011 | 0.75 | 0.037 | <i>I</i> | 0.005 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+UPO | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.85 | 0.011 | 0.75 | 0.037 | <i>I</i> | 0.005 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.85 | 0.011 | 0.75 | 0.037 | <i>I</i> | 0.005 | <i>I</i> | 0.002 | <i>I</i> |
| S7 | FOT+R | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 | 0.037 | 0.75 | 0.037 | 0.8 | 0.019 | 0.95 | 0.007 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 |
| | COT+UPO | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 | 0.037 | 0.75 | 0.037 | 0.8 | 0.019 | 0.95 | 0.007 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 |
| | COT+R | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 | 0.037 | 0.75 | 0.037 | 0.8 | 0.019 | 0.95 | 0.007 | 0.95 | 0.003 | 0.95 | 0.003 | 0.85 | 0.011 | 0.95 | 0.003 | 0.8 | 0.019 | 0.75 |
| S8 | FOT+R | 0.99 | 0.008 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.99 | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.004 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | 0.99 | 0.008 | <i>I</i> | 0.002 | 0.99 |
| | COT+UPO | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| S9 | FOT+R | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | <i>I</i> | 0.005 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+UPO | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |
| | COT+R | <i>I</i> | 0.004 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | <i>I</i> | 0.005 | 0.95 | 0.003 | 0.9 | 0.006 | 0.85 | 0.011 | <i>I</i> | 0.004 | <i>I</i> | 0.002 | <i>I</i> |

Results that are statistically significant are in italicized form.

Table 6 Percentage of faults detected by each testing approach

| Setting | Approach | ArduCopter | | | | | ArduPlane | | ArduRover | | Avg. (%) |
|---------|----------|------------|---------|---------|---------|---------|-----------|---------|-----------|---------|----------|
| | | SH1 (%) | SH2 (%) | SH3 (%) | SH4 (%) | SH5 (%) | SH1 (%) | SH2 (%) | SH1 (%) | SH2 (%) | |
| S1 | FOT+UPO | 10 | 10 | 10 | 8 | 7 | 35 | 40 | 30 | 20 | 19 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S2 | FOT+UPO | 43 | 45 | 50 | 43 | 60 | 95 | 100 | 100 | 100 | 71 |
| | FOT+R | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 20 | 4 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S3 | FOT+UPO | 70 | 60 | 67 | 55 | 70 | 95 | 100 | 100 | 100 | 80 |
| | FOT+R | 3 | 0 | 0 | 0 | 0 | 5 | 10 | 20 | 40 | 9 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S4 | FOT+UPO | 17 | 15 | 10 | 17 | 17 | 55 | 80 | 90 | 90 | 43 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S5 | FOT+UPO | 40 | 48 | 48 | 70 | 67 | 100 | 100 | 100 | 100 | 75 |
| | FOT+R | 3 | 0 | 0 | 3 | 0 | 10 | 10 | 10 | 30 | 7 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S6 | FOT+UPO | 77 | 65 | 63 | 77 | 80 | 100 | 100 | 100 | 100 | 85 |
| | FOT+R | 10 | 0 | 3 | 3 | 0 | 15 | 10 | 30 | 50 | 13 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 2 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S7 | FOT+UPO | 30 | 15 | 13 | 13 | 15 | 65 | 90 | 100 | 100 | 49 |
| | FOT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S8 | FOT+UPO | 67 | 63 | 58 | 63 | 68 | 100 | 100 | 100 | 100 | 80 |
| | FOT+R | 3 | 0 | 3 | 0 | 3 | 15 | 10 | 20 | 30 | 9 |
| | COT+UPO | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S9 | FOT+UPO | 93 | 73 | 78 | 70 | 75 | 100 | 100 | 100 | 100 | 88 |
| | FOT+R | 10 | 3 | 5 | 0 | 3 | 20 | 20 | 40 | 50 | 17 |
| | COT+UPO | 0 | 3 | 0 | 0 | 0 | 5 | 0 | 10 | 10 | 3 |
| | COT+R | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 1 |

incorrectly adjust its orientation, leading to a collision. Therefore, it is necessary to test self-healing behaviors in the presence of environmental uncertainties. To build such a testing environment, we adapt the software in the loop approach. In this approach, uncertainties are explicitly introduced via sensor data and actuation instructions. Second, it requires a specific sequence of operation invocations and a specific sequence of uncertainty values to reveal a fault caused by the effect of uncertainties. Invoking different operations or invoking the same operation with different inputs can both lead to distinct system states. Moreover, the impacts of uncertainties cause the states to diverge further. Since a fault may only be activated in a few special states, specific operation invocations and uncertainty values need to be found to reveal the fault. In this context, coverage-oriented testing, which aims to evenly explore each system state, is ineffective to find faults. To address this issue, we present a fragility-oriented approach in this paper. By focusing on the fragile states of the SUT, it managed to find faults more effectively. Third, FOT and UPO have to cooperate to effectively detect faults under uncertainties. Directed by the fragility obtained from execution, FOT and UPO can gradually learn

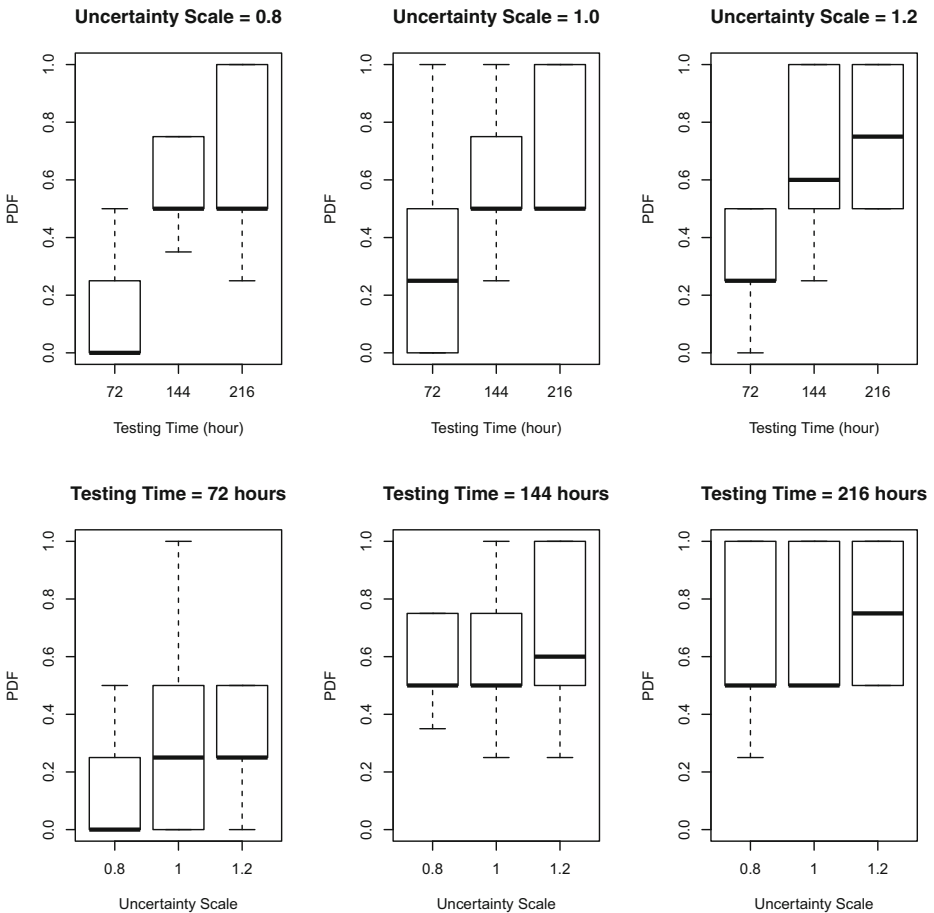


Fig. 10 Box plots of PDF under each testing time and uncertainty scale

the optimal policy to select operation invocations and the optimal uncertainty generation policy, respectively. The experiment results demonstrated that compared with the other approaches, FOT+UPO could enhance the fault detection ability by at least 50%.

8.5 Threats to validity

Conclusion validity is concerned with factors that affect the conclusion drawn from the outcome of experiments (Wohlin et al. 2012). Because of random transition selection and random uncertainty generation used by the four testing approaches, randomness in the results is the most probable conclusion validity threat. To reduce this threat, all the testing jobs were repeated ten times. We applied the Mann–Whitney U test and Vargha and Delaney's \hat{A}_{12} to evaluate the statistical difference and magnitude of improvement.

Internal validity threat refers to the influence that affects the causal relationship between the treatment and outcome (Wohlin et al. 2012), i.e., the testing approach and its fault detection ability. Since testing time and scale of uncertainty have impacts on the performance of a testing approach, they could be the threat to internal validity. To reduce such threat, we compared the

performance of the four selected testing approaches under three testing times and three uncertainty scales.

External validity threats concern the generalization of the experiment results (Wohlin et al. 2012). We employed nine self-healing behaviors from three real case studies to compare the performance of four testing approaches. However, additional case studies are needed to generalize the results further.

Construct validity is concerned with how well the metrics used in the experiment reflect the construct (Wohlin et al. 2012)—fault detection ability of a testing approach. Because the number of actual faults is unknown, we used the number of detected faults and the percentage of detected fault as the evaluation metrics, which are comparable across the four testing approaches.

9 Related work

This section presents the related work from three aspects: model-based testing in Section 9.1, testing with reinforcement learning in Section 9.2, and uncertainty-wise testing in Section 9.3.

9.1 Model-based testing

Model-based testing (MBT) has shown good results of producing effective test suites to reveal faults (Enoiu et al. 2016). For a typical MBT approach, abstract test cases are generated from models first, e.g., using structural coverage criteria (e.g., all state coverage) (Utting et al. 2012; Grieskamp et al. 2011). Generated abstract test cases are then transformed into executable ones, which are executed on the SUT. To reduce the overhead caused by test cases generation, researchers proposed to combine test generation, selection, and execution into one process (de Vries and Tretmans 2000; Larsen et al. 2004). de Vries and Tretmans (2000) created a testing framework, with which the SUT is modeled as a labeled transition system. By parsing this model, test inputs are generated on the fly to perform conformance testing. This approach aims to test all paths belonging to this model. However, if loops exist or the specified model is large, additional mechanisms are required to reduce the state space. Larsen et al. (2004) proposed a similar testing tool for embedded real-time systems. It uses the timed I/O transition system as the test model, and test inputs are randomly generated from the model on the fly for testing.

Different from the existing works, the proposed fragility-oriented testing approach relies on the execution of ETMs to facilitate the testing of SH-CPSs under uncertainty. During the execution, FOT and UPO apply reinforcement learning techniques to learn the optimal policy of invoking operations and best policy of generating uncertainties, respectively. In addition, our work focuses on testing self-healing behaviors in the presence of environmental uncertainty, which is not covered by existing works.

9.2 Testing with reinforcement learning

The first reinforcement learning-based testing algorithm was proposed in Veanes et al. (2006). It uses frequencies of transitions' coverage as the heuristics of reinforcement learning. Directed by the frequencies, the algorithm tries to explore all transitions equally. However, a long-term reward is not realized in this approach. Groce et al. Groce et al. (2012) created a framework to simplify the application of reinforcement learning

for testing, which uses coverage as the heuristic and relies on SARSA(λ) (Sutton and Barto 1998) for calculating long-term rewards. Similarly, Araiza-Illan et al. (2016) used coverage as the reward function to test human–robot interactions with reinforcement learning. Due to uncertainty, achieving the full transition coverage is insufficient to find faults in self-healing behaviors. Thus, we propose to use fragility instead of coverage as the heuristic. Moreover, we devised two novel algorithms, i.e., FOT and UPO, for operation invocation and uncertainty generation, respectively.

9.3 Uncertainty-wise testing

Regarding uncertainty-wise testing, some taxonomies of uncertainty for self-adaptive systems have been proposed in Ramirez et al. (2012) and Esfahani and Malek (2013), and a conceptual model of uncertainty for CPSs has been built in Zhang et al. (2015). To test systems in the presence of the uncertainty, Fredericks et al. (2014) developed a run-time testing framework. It dynamically adapts a set of predefined test cases to test whether the SUT behaves correctly when adaptation is required to handle changes in environmental conditions. However, the paper does not mention how to obtain the initial test cases and how to construct an uncertainty-introduced testing environment. Yang et al. (2014) devised a formal approach to verify the correctness of self-adaptive applications under uncertainty, while the formal verification approach is computationally expensive, and it requires extra effort to prove the SUT is consistent with the verified model. Zhang et al. (2017) proposed a multi-objective search-based approach for test case generation and minimization, with the aim of discovering unknown uncertainties.

Different from the existing works, we aim to test whether the SUT can behave properly in the presence of uncertainty. To effectively detect faults, we devise the UPO algorithm. By utilizing the fragility to optimize the uncertainty generation policy (an ANN), it manages to effectively find a sequence of uncertainty values that can cooperate with a sequence of operation invocations to reveal faults.

10 Conclusion

This paper presents a fragility-oriented approach for testing self-healing cyber-physical systems (SH-CPSs) under uncertainty. The testing approach consists of two steps. One is to select a sequence of operation invocations, which determines the behavior of the SH-CPS under testing (SUT) in test execution. The other is to generate a sequence of uncertainty values to make the SUT behave under uncertainty. For the two steps, we devise two algorithms: fragility-oriented testing (FOT) and uncertainty policy optimization (UPO). Both of them employ the fragility to learn the optimal policies for operation invocations and uncertainty generation, respectively. To evaluate their performance, we compared them against three testing approaches: FOT+R, COT+UPO, and COT+R, where COT represents a coverage-oriented algorithm for operation invocations and R represents a random mechanism for uncertainty generation. The four testing approaches were applied to test nine self-healing behaviors from three real-world case studies. The testing results showed that FOT+UPO significantly detected more faults than the other three approaches, in 73 out of 81 testing jobs. In the 73 jobs, FOT+UPO detected more than 70% of faults, while the others detected 17% of faults, at the most.

Funding information This research is funded by the Research Council of Norway (RCN) under MBT4CPS project (grant no. 240013/O70). Tao Yue and Shaukat Ali are also supported by the RCN funded Zen-Configurator project (grant no. 240024/F20), RFF Hovedstaden funded MBE-CR project (grant no number. 239063), and Certus SFI and EU Horizon 2020 funded U-Test project (grant no. 645463).

Appendix

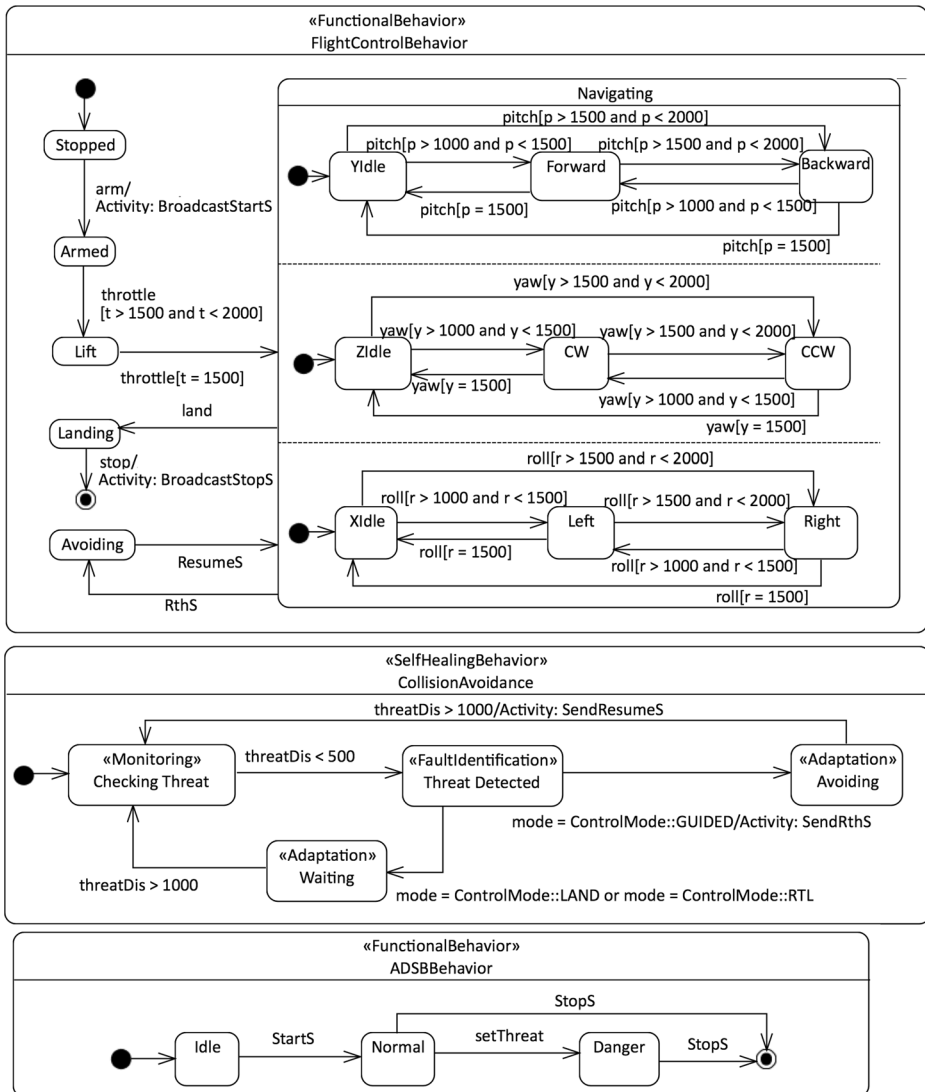


Fig. 11 Simplified ETM for ArduCopter. Figure 11 presents a simplified ETM for ArduCopter. According to the ETM, a DFSM can be constructed, and part of the DFSM is shown in Fig. 5

References

- Ali, S., Iqbal, M. Z., Arcuri, A., & Briand, L. C. (2013). Generating test data from OCL constraints with search techniques. *IEEE Transactions on Software Engineering*, 39(10), 1376–1402.
- Anzai, Y. (2012). *Pattern recognition and machine learning*. New York: Elsevier.
- Araiza-Illan, D., Pipe, A. G., & Eder, K. (2016). Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In *Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering*. ACM.
- Arcuri, A., & Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd International Conference on Software Engineering (ICSE)* (pp. 1–10). IEEE.
- Aronson, J. E., Liang, T.-P., & Turban, E. (2005). *Decision support systems and intelligent systems*. Pearson Prentice-Hall.
- Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. arXiv preprint arXiv:1708.05866.
- Asadollah, S. A., Inam, R., & Hansson, H. (2015). A survey on testing for cyber physical system. In *The 27th IFIP International Conference on Testing Software and Systems* (pp. 194–207). Sharjah and Dubai: Springer.
- Bures, T., Weyns, D., Berger, C., Biffi, S., Daun, M., Gabor, T., et al. (2015). Software engineering for smart cyber-physical systems—towards a research agenda: report on the First International Workshop on Software Engineering for Smart CPS. *ACM SIGSOFT Software Engineering Notes*, 40(6), 28–32.
- de Vries, R. G., & Tretmans, J. (2000). On-the-fly conformance testing using SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 382–393.
- Demuth, B., & Wilke, C. (2009). Model and object verification by using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia* (pp. 687–690). Citeseer.
- Duan, Y., Chen, X., Houthoofd, R., Schulman, J., & Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning* (pp. 1329–1338).
- Enoiu, E. P., Caeuic, A., Sundmark, D., & Pettersson, P. (2016). A controlled experiment in testing of safety-critical embedded software. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1–11). IEEE.
- Esfahani, N., & Malek, S. (2013). Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II* (pp. 214–238). Springer.
- Fredericks, E. M., DeVries, B., & Cheng, B. H. (2014). Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (pp. 17–26). ACM.
- Glynn, P. W., & Iglehart, D. L. (1989). Importance sampling for stochastic simulations. *Management Science*, 35(11), 1367–1392.
- Grieskamp, W., Hierons, R. M., & Pretschner, A. (2011). Model-based testing in practice. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., et al. (2012). Lightweight automated testing with adaptation-based programming. In *IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)* (pp. 161–170). IEEE.
- Hestenes, M. R., & Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards, NIST*, 49, 409–435.
- Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4, 237–285.
- Kakade, S., & Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *The Nineteenth International Conference on Machine Learning (ICML)* (pp. 267–274). PMLR.
- Larsen, K. G., Mikucionis, M., & Nielsen, B. (2004). Online testing of real-time systems using uppaal. In *International Workshop on Formal Approaches to Software Testing* (pp. 79–94). Springer.
- Li, Y. (2017). Deep reinforcement learning: an overview. arXiv preprint arXiv:1701.07274.
- Ma, T., Ali, S., Yue, T., & Elaasar, M. (2017). Fragility-oriented testing with model execution and reinforcement learning. In *IFIP International Conference on Testing Software and Systems* (pp. 3–20). Springer.
- Ma, T., Ali, S., & Yue, T. (2018). Modeling foundations for executable model-based testing of self-healing cyber-physical systems. In *International Journal on Software and Systems Modeling (SoSyM)* (pp. 1–31). Berlin, Heidelberg: Springer.
- Mooney, C. Z., Duval, R. D., & Duval, R. (1993). *Bootstrapping: a nonparametric approach to statistical inference* (No. 95). Newbury Park: Sage.
- Ramirez, A. J., Jensen, A. C., & Cheng, B. H. (2012). A taxonomy of uncertainty for dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (pp. 99–108). IEEE.

- Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)* (pp. 1889–1897). PMLR.
- Schupp, S., Ábrahám, E., Chen, X., Makhlof, I. B., Frehse, G., Sankaranarayanan, S., et al. (2015). Current challenges in the verification of hybrid systems. In *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems* (pp. 8–24). Springer.
- Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: an introduction* (1st edn.). Cambridge: MIT Press.
- Tatibouet, J. (2016). Moka—a simulation platform for Papyrus based on OMG specifications for executable UML. In *2016 EclipseCon Europe Conference*. Ludwigsburg: OSGI.
- Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5), 297–312.
- Veanes, M., Roy, P., & Campbell, C. (2006). Online testing with reinforcement learning. In *Formal Approaches to Software Testing and Runtime Verification*. 240–253.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering* (1st edn.). New York: Springer Science & Business Media.
- Yang, W., Xu, C., Liu, Y., Cao, C., Ma, X., & Lu, J. (2014). Verifying self-adaptive applications suffering uncertainty. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (pp. 199–210). ACM.
- Yegnanarayana, B. (2009). *Artificial neural networks* (1st edn.). New Delhi: PHI Learning Pvt. Ltd.
- Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., & Norgren, R. (2015). Understanding uncertainty in cyber-physical systems: a conceptual model. In *Modelling Foundations and Applications: 12th European Conference*. ECMFA Springer.
- Zhang, M., Ali, S., & Yue, T. (2017). *Uncertainty-wise test case generation and minimization for cyber-physical systems: a multi-objective search-based approach*. Simula. Technical report: <https://www.simula.no/publications/uncertainty-based-test-case-generation-and-minimization-cyber-physical-systems-multi>.
- Zheng, X., & Julien, C. (2015). Verification and validation in cyber physical systems: research challenges and a way forward. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems* (pp. 15–18). IEEE Press.

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 currently a senior research scientist in Simula Research Laboratory, Norway. His research focuses on devising novel methods for verification and validation (V&V) of large-scale highly connected software-based systems that are commonly referred to as cyber-physical systems (CPSs). He has been involved in several basic researches, research-based innovation, and innovation projects in the capacity of PI/Co-PI related to model-based testing (MBT), search-based software engineering, and model-based system engineering. He has rich experience of working in several countries including UK, Canada, Norway, and Pakistan. Shaukat has been on the program committees of several international conferences (e.g., MODELS, ICST, GECCO, SSBSE) and also served as a reviewer for several software engineering journals (e.g., *TSE*, *IST*, *SOSYM*, *JSS*, *TEVC*). He is also actively participating in defining international standards on software modeling in Object Management Group (OMG), notably a new standard on uncertainty modeling.



Tao Yue is a chief research scientist at Simula Research Laboratory, Oslo, Norway, and the Head of the Department of Engineering Complex Software Systems. She is also affiliated with the University of Oslo. She has received the Ph.D. degree in the Department of Systems and Computer Engineering at Carleton University, Ottawa, Canada, in 2010. Before that, she was an aviation engineer and system engineer for 7 years. She has nearly 20 years of experience of conducting industry-oriented research with a focus on model-based engineering (MBE) in various application domains such as Avionics, 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 requirements engineering, MBE, model-based testing, uncertainty-wise testing, uncertainty modeling, search-based software engineering, empirical software engineering, and product line engineering, with a particular focus on large-scale software systems such as cyber-physical systems. Dr. Yue has been on the program and organization committees of several international conferences (e.g., MODELS, RE, SPLC). She is also on the editorial board of *Empirical Software Engineering* and *Science of Computer Programming*. Dr. Yue is also actively participating in defining international standards in Object Management Group (OMG), including precise semantics for uncertainty modeling.



Maged Elaasar is a senior software architect and consultant. He has earned his Ph.D. in Electrical and Computer Engineering (2012) and his M.Sc. in Computer Science (2003) from Carleton University. He is the founder of Modelware Solutions, a company that provides development, consulting, and training services in the area of model-based engineering (MBE). Maged's expertise in MBE spans both technology and practice. On the technology side, Maged is now leading the development of CAESAR, an integrated suite of software tools that streamlines the practice of model-based systems engineering (MBSE) for large projects. He has also led the development of Rational Software Architect, a leading commercial UML modeling tool. Maged has also been an active contributor to modeling-related open-source projects at Eclipse, like GMF Runtime and Papyrus. He has also led modeling-related open standards at OMG, like UML (co-chair), SysML, and DD (co-chair). On the practice side, Maged has been a systems architect modeling the architecture of complex space projects in SysML. He has also co-developed an executable systems engineering method (ESEM) that supports automated systems analysis using model execution.