Fragility-Oriented Testing with Model Execution and Reinforcement Learning

Tao Ma, Shaukat Ali, Tao Yue, Maged Elaasar

Fragility-Oriented Testing with Model Execution and Reinforcement Learning[·]

Tao Ma[1], Shaukat Ali[1],Tao Yue[1,2], Maged Elaasar[3]

[1]Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
[2]University of Oslo, P.O. Box 1072, 0316 Blindern, Norway
{taoma, shaukat, tao}@simula.no
[3] Carleton University, 1125 Colonel By Dr., Ottawa, ON, K1S5B6, Canada
melaasar@gmail.com

Abstract. Self-healing is becoming an essential behavior of smart Cyber-Physical Systems (CPSs), which enables them to recover from faults by themselves. Such behaviors make decisions autonomously at runtime and they often operate in an uncertain physical environment making testing even more challenging. To this end, we propose Fragility-Oriented Testing (FOT), which relies on model execution and reinforcement learning to cost-effectively test self-healing behaviors of CPSs in the presence of environmental uncertainty. We evaluated FOT's performance by comparing it with a Coverage-Oriented Testing (COT) algorithm. Evaluation results show that FOT significantly outperformed COT for testing nine self-healing behaviors implemented in three case studies. On average, FOT managed to find 80% more faults than COT and for cases when both FOT and COT found the same faults, FOT took on average 50% less time than COT.

Keywords: Cyber-Physical Systems, Uncertainty, Self-Healing Behaviors, Model Execution, Reinforcement Learning

1. Introduction

Self-healing is becoming an important functionality of smart Cyber-Physical Systems (CPSs) [1]. With such functionality, a Self-Healing CPS (SH-CPS) has the ability to recover from faults and adapt its behavior accordingly. Given that uncertainty is inherent in CPSs since such systems operate in highly unpredictable physical environment [2], self-healing behaviors of an SH-CPS must deal with uncertainty gracefully. By uncertainty, we mean *"the lack of knowledge of which value an uncertain factor will take at a given point of time during execution"* [3]. In this paper, we limit our scope to uncertain factors related to sensing (e.g., noise) and actuation (e.g. deviation) of SH-CPSs as a starting point.

To check the correctness of self-healing behaviors of SH-CPSs in the presence of uncertainty, a cost-effective testing method is required. In this paper, we propose

[·] This research was funded by the MBT4CPS project (grant no. 240013/O70).

a Fragility-Oriented Testing (FOT) algorithm to ensure that self-healing behaviors properly handle environmental uncertainty. The core idea of FOT is using *fragility* (i.e., a measure indicating how near an SH-CPS is to fail in a given state) as a heuristic for revealing faults. Based on this heuristic, we use model execution and reinforcement learning to explore various execution paths of the SH-CPS and simulate uncertainty in its physical environment to cost-effectively find faults.

A traditional model-based testing (MBT) approach generates test cases from test models with a test strategy and executes them on a system in separate steps. In contrast, FOT tests a system with a test strategy that dynamically incorporates information during execution to decide the next test execution step. From the execution information, FOT uses a reinforcement learning method to identify transitions that have high possibilities to reveal a fault, i.e., lead to a state with the highest fragility. Accordingly, FOT adapts its transition selection policy to favor these transitions for test execution with the aim to cost-effectively find faults.

We evaluated FOT by testing nine self-healing behaviors in three case studies. We conducted one experiment per self-healing behavior and compared cost (measured as time to find a fault) and effectiveness (measured as the number of faults found) of FOT as opposed to Coverage-Oriented Testing (COT) [4]. Each self-healing behavior was tested under 10 environmental uncertainties. Evaluation results show that FOT significantly outperformed COT in five out of nine experiments in terms of finding faults. On average, FOT found 80% more faults and spent 50% less test execution time to find a fault than COT. Note that COT was used as a naive baseline to be compared with FOT and comparison with more sophisticated comparable algorithms is required in the future.

Our key contributions: 1) proposing a reinforcement learning based testing algorithm to cost-effectively find faults in SH-CPSs under uncertainty, 2) defining *fragility* as the heuristic to guide the reinforcement learning algorithm, 3) evaluating FOT (by comparing with COT) in terms of cost-effectiveness for testing nine self-healing behaviors implemented in three real case studies. We organize the paper as follows. Section 2 presents the background, Section 3 presents the running example, and the FOT is presented in Section 4. Section 5 presents an evaluation, Section 6 summarizes related work, and Section 7 concludes the paper.

2. Background

This section discusses Executable Test Model (ETM) and Dynamic Flat State Machine (DFSM), the key models used in FOT, in Section 2.1 and Section 2.2. Section 2.3 briefly summarizes a test model execution framework – TM-Executor.

2.1 Executable Test Model (ETM)

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, an SH-CPS monitors fault occurrences and adapts its behavior if a fault is detected with self-healing behaviors. 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 [3], which allows creating an Executable Test Model (ETM) for an 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: $SM = \{sm_1, \ldots, sm_n\}$, where each state machine sm_i has MoSH stereotype applied. A sm_i has a set of states $S_{sm_i} = \{s_{sm_i1}, \ldots, s_{sm_is}\}$ and transitions $T_{sm_i} = \{t_{sm_i1}, \ldots, t_{sm_it}\}$. 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 a constraint in OCL[1] constraining one or more state variables. When s_{sm_ij} is active, its corresponding state invariant should 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 API 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 fUML[3] and Precise Semantics Of UML State Machines(PSSM)[4] standards, the specified state machines are executable. Thus the test model is called an Executable Test Model.

2.2 Dynamic Flat State Machine (DFSM)

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 DFSM during test execution. A DFSM has a set of states $\mathbb{S} = \{\mathbb{s}_1, \mathbb{s}_2, \ldots, \mathbb{s}_n\}$ and a set of transitions $\mathbb{T} = \{\mathbb{t}_1, \mathbb{t}_2, \ldots \mathbb{t}_m\}$. Each state $\mathbb{s}_i$ in $\mathbb{S}$ is constituted by states s_{sm_ij} from each sm_i, denoted as $\mathbb{s}_i = s_{sm_1i} \wedge s_{sm_2j} \wedge \ldots \wedge s_{sm_nk}$. Accordingly, the conjunction of all constituents' state invariants $[o_{sm_1i} \wedge o_{sm_2j} \wedge \ldots \wedge o_{sm_nk}]$ forms the state invariant of $\mathbb{s}_i$, denoted as $\mathbb{o}_i$. Meanwhile, the set of transitions connecting the DFSM states is captured by $\mathbb{T}$. Each transition $\mathbb{t}_i$ belonging to $\mathbb{T}$ is uniquely mapped to a transition t_{sm_xj} in a state machine sm_x, expressed as $\mathbb{t}_i = t_{sm_xj}$. While the ETM is being executed, the DFSM of the ETM is dynamically constructed. FOT uses the DFSM to learn the value of firing each transition and find the optimal transition selection policy to cost-effectively find faults. Thus, we mainly use DFSM to explain FOT.

2.3 Test Model Execution Framework

We developed a testing framework called TM-Executor [3] in our previous work,

[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

which executes the ETM and the SUT at the same time. Via testing APIs, state variable values are queried from the SUT and used by TM-Executor to evaluate state invariants of the active state. If an invariant is evaluated false, it means that the SUT fails to behave consistently with the ETM and a fault is detected.

The execution of an ETM results in the execution of the SUT. During the execution, TM-Executor dynamically and incrementally derives a DFSM from the set of concurrent state machines in the ETM. As aforementioned, a transition's trigger op and guard g specify which operation to invoke with which input parameter values to make the SUT and the ETM transit from one state to another. While an operation is being invoked, an operation call event is generated, which drives the execution of the ETM. Meanwhile, the operation is executed to call a corresponding testing API, which makes the SUT enter the next state.

Two kinds of testing APIs for controlling the SUT can be specified as a transition's trigger op. One is functional control operation, which instructs the SUT to execute a nominal functional operation. Second 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.

3. Running Example

We will use a running example of an Unmanned Aerial Vehicle control system (i.e., ArduCopter[5]) to illustrate FOT. It has two physical units, i.e., copter and Ground Control Station (GCS). With the GCS, users remotely control the copter using a number of flight modes. During the flight, the copter is constantly affected by environmental uncertainties such as wind speed and direction, measurements noise from the GPS, accelerometer, and compass. This poses an extra challenge to the self-healing behaviors of the copter. *Collision avoidance* 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. We build an ETM to specify the expected collision avoidance behavior along with related functional behaviors. Fig. 1 presents a partial simplified DFSM corresponding to the ETM; while, the complete ETM is presented in [5]. We take one path (bold transitions in Fig. 1: $\mathbb{t}1 \rightarrow \mathbb{t}2 \rightarrow \mathbb{t}3 \rightarrow \mathbb{t}4 \rightarrow \mathbb{t}11 \rightarrow \mathbb{t}12 \rightarrow \mathbb{t}16 \rightarrow \mathbb{t}18$) to explain test execution.

Starting from the *Initial* state, the DFSM directly enters *Stopped*, as there is no trigger on $\mathbb{t}1$. From *Stopped*, TM-Executor fires $\mathbb{t}2$ by calling the functional control operation *start* to launch the SUT. As a result, *Started* becomes active. To make the copter enter state *Lift*, TM-Executor invokes operation *throttle* with a valid value of input parameter *thr* obtained by solving guard constraint *[thr > 1600 and thr < 2000]* via constraint solver EsOCL [6]. Then, the copter takes off and reaches the *Lift* state. In the *Lift* state, TM-Executor needs to choose one of the two outgoing transitions to be triggered. Assuming $\mathbb{t}4$ is chosen, it is triggered by invoking *pitch* with a valid value of *pit* satisfying *[pit > 1000 and pit < 1400]*. This invocation triggers the copter to move forward. In the *Forward* state, TM-Executor either

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

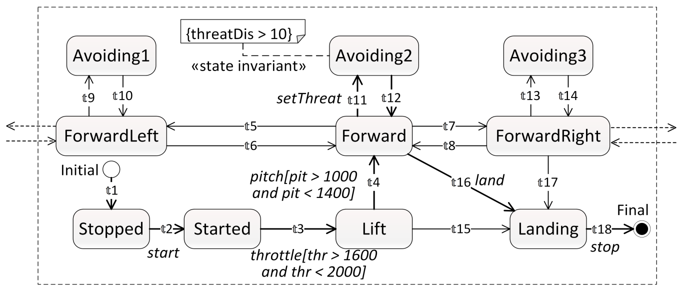

Fig. 1 A Simplified DFSM for ArduCopter

changes the copter's movement (i.e., firing 𝕥5, 𝕥7, or 𝕥16) or invokes the fault injection operation *setThreat*, which simulates that an aircraft is approaching from the left behind of the copter to trigger the collision avoidance behavior. Here the second option is adopted. Triggered by this, the collision avoidance behavior controls the copter to fly away from the aircraft. When the distance between them (*threatDis*) is over 1000 meters (not shown in Fig. 1), the collision threat is avoided and the copter's flight mode changes back to the previous one. Hence, 𝕥12 is traversed[6]. Then TM-Executor chooses to trigger 𝕥16, followed by firing 𝕥18, to stimulate that the copter passes through the *Landing* state and reaches the final state.

In parallel to the execution, TM-Executor periodically obtains the values of the SUT's state variables through testing APIs and repeatedly uses these values to evaluate the active state's invariant, using a constraint evaluator DresdenOCL [7]. If an invariant is evaluated to be false, then a fault is detected.

The decision to which transition to be triggered determines if a fault can be found by executing the ETM and the SUT. 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., 𝕥9 must be activated. However, activating 𝕥9 once may not be sufficient to find the fault. One reason is that a large number of input parameter values could be used to invoke an operation for firing a transition, e.g., 𝕥4 (Fig. 1). Each input leads to a distinct flight orientation and only in particular situations, the collision is likely to happen. Another reason is the effect of uncertainties. Measurement uncertainties from sensors and actuation offset from actuators change from time to time. Different values of the uncertain factors[7] lead

[6] When a collision is avoided, the copter is back to the flight mode. Hence, no testing API needs to be invoked to trigger 𝕥12. 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.

[7] Uncertain factor is a feature (e.g., a parameter) whose value is uncertain due to lack of knowledge.

to diverse orientations, making it even harder to reveal the fault.

Therefore, TM-Executor mainly identifies which transition in the DFSM will most likely reveal a fault and frequently triggers the transition to reveal the fault. Since the DFSM is normally large, containing hundreds or thousands of transitions, fulfilling this task is non-trivial. We therefore present a novel cost-effective testing algorithm, FOT. It uses a reinforcement learning method to learn values of firing each transition, which helps TM-Executor to cost-effectively find faults.

4. Fragility-Oriented Testing under Uncertainty

In this section, we present details of the testing algorithm FOT (Section 4.1) and three uncertainty generation strategies (Section 4.2), which together enable TM-Executor to cost-effectively find faults in the SUT under uncertainty. Section 4.3 describes the implementation of FOT and the uncertainty generation strategies.

4.1 Testing Algorithm

Definition 1. The fragility of the SUT in a given state expressed as $F(\mathbb{s})$, is a real value between 0 and 1. It describes how close (distance wise) the state invariant of $\mathbb{s}$ 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 $F(\mathbb{s})$ as follows:

$$F(\mathbb{s}) = 1 - dis(\neg \mathbb{O}) \tag{1}$$

where $\neg \mathbb{O}$ is the negation of state $\mathbb{s}$'s invariant $\mathbb{O}$ and $dis(\neg \mathbb{O})$ is a distance function (adopted from [6]) that returns a value between 0 and 1 indicating how close the constraint $\neg \mathbb{O}$ 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 $dis\big(\neg(threatDis\ >\ 10)\big) = \frac{(15-10)+1}{(15-10)+1+1} = 0.86$[8]. 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, $1 - dis(\neg \mathbb{O})$ is used to define the fragility of the SUT in state $\mathbb{s}$.

Definition 2. The T-value of a transition expressed as $T(\mathbb{t})$, is a real value between 0 and 1. It states the possibility that a fault can be revealed after firing the transition. 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:

$$T(\mathbb{t}) = \max_{\mathbb{s} \in \mathbb{S}_{next}} \{\gamma^n \cdot F(\mathbb{s})\} \tag{2}$$

where γ $(0 \leq \gamma < 1)$ is a discount rate; n is the number of transitions between $\mathbb{s}$ and $\mathbb{t}$'s target state; and $\mathbb{S}_{next}$ is a set of states that can be reached from $\mathbb{t}$'s source state via a path in the DFSM. 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 $\mathbb{t}$'s target state. For example, in

[8] The distance function of greater operator is: $dis(x > y) = (y - x + k)/(y - x + k + 1)$ $, when\ x \leq y$, where k is an arbitrary positive value. Here we set k=1. More details are in .

Fig. 1, to obtain the T-value of $\mathbb{t}4$, we calculate the discounted fragility of each state in $\mathbb{S}_{next}$. For the fragility of *Avoiding1*, it needs to be discounted by γ^2, since at least two transitions $\mathbb{t}5$ and $\mathbb{t}9$ are required to connect the *Avoiding1* state to $\mathbb{t}4$'s target state *Forward*. Clearly, when γ equals 0, only the fragility of $\mathbb{t}$'s target state is considered. While, γ approaching 1 makes a state to be reached more important.

Overview. The objective of FOT is to find the optimal transition selection policy to cost-effectively find faults. 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 cost-effectively find faults. The pseudocode of FOT is presented below with in total 17 lines (L1-L17).

At the beginning, all transitions' T-values are unknown. As every transition has a possibility to reveal a fault, we initialize an estimated T-value of each transition with the highest one (L1, L2). This encourages the algorithm to extensively explore uncovered transitions. After that, iterations of test execution and the learning process begin. At each iteration, the execution of the ETM 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 continuous calculation of T-values. Whenever, the SUT enters a state $\mathbb{s}$, FOT selects one of the outgoing transitions of $\mathbb{s}$ 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 $\mathbb{s}$ to $\mathbb{s}'$. If the state invariant of $\mathbb{s}'$ is not satisfied, then a fault is detected (L12 - L15). Otherwise, FOT evaluates the fragility of the SUT in $\mathbb{s}'$ (L16), i.e., $\mathbb{F}(\mathbb{s}')$, and uses $\mathbb{F}(\mathbb{s}')$ to update estimated T-values. Since it is possible to reach $\mathbb{s}'$ via numerous transitions, finding all these transitions and

```
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 ← 1          // initialize T-values of transitions
        3 for i=1 to maxIteration
        4   etm.Start( )
        5   while etm.ReachFinalState( ) is false
        6     dfsm ← EnrichDFSM(etm)          // dynamically construct the DFSM
        7     reachedTransitions ← dfsm.activeState.outgoingTransitions
        8     selectedTransition ← SoftmaxSelect(reachedTransitions) //select transition
        9     executor.Trigger(selectedTransition)
       11     stateInvariant ← selectedTransition.target.invariant
       12     if executor.Evaluate(stateInvariant) is false
       13       LogFaultDetected(selectedTransition)
       14       dfsm.Remove(selectedTransition)
       15       break
       16     fragility ← executor.DistanceToViolation(stateInvariant)
       17     executor.UpdateTvalue(selectedTransition, fragility)
                                             // revise the T-value of selectedTransition
  End
```

updating their T-values are computationally impractical for an ETM with hundreds of transitions. Thus FOT only updates the estimated T-value of the last triggered transition (L17). Since $\mathbb{F}(\mathbb{s}^{'})$ is not a constant value, the upper bound of $\mathbb{F}(\mathbb{s}^{'})$ 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 true values. In this way, the T-values are learned from the execution and the learned T-values direct FOT to cost-effectively find faults. Note that testing budget determines the maximum number of iterations. If it is too small, FOT may not able to find faults. The details of T-value learning and transition selection policy are explained next.

T-value Learning. Before executing the SUT and the ETM, the T-value $\boldsymbol{T}(\mathbb{t})$ of every transition is unknown. We adopt a reinforcement learning approach to learn $\boldsymbol{T}(\mathbb{t})$ from execution. A fundamental property of $\boldsymbol{T}(\mathbb{t})$ is that it satisfies a recursive relation, which is called the Bellman Equation [8], as shown in the formula below:
Recursive relation between $T(\mathbb{t})$ and $T(\mathbb{t}_{suc})$:

$$T(\mathbb{t}) = \max\{F(\mathbb{s}_{tar}), \gamma \cdot \max_{\mathbb{t}_{suc} \in \mathbb{T}_{suc}} T(\mathbb{t}_{suc})\} \tag{3}$$

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

Inspired by Q-learning [8], a reinforcement learning method, FOT uses the estimated T-value $ET(\mathbb{t})$ to approximate $T(\mathbb{t})$, i.e., the true T-value. $ET(\mathbb{t})$ is updated in the following way to make it approach $T(\mathbb{t})$.

$$ET(\mathbb{t})^{'} = max\{F(\mathbb{s}_{tar}), \gamma \cdot \max_{\mathbb{t}_{suc} \in \mathbb{T}_{suc}} ET(\mathbb{t}_{suc})\} \tag{4}$$

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

Equation (4) enables FOT to iteratively update $ET(\mathbb{t})$. Once a transition $\mathbb{t}$ is triggered, the fragility of the SUT in $\mathbb{t}$'s target state $F(\mathbb{s}_{tar})$ can be evaluated using Equation (1). Using Equation (4), $ET(\mathbb{t})$ can be updated whenever a fragility is obtained. As proved in [8], as long as the estimated T-values are continuously updated, $ET(\mathbb{t})$ will converge to the true T-value: $T(\mathbb{t})$.

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 [9] to predict the distribution of the fragility and select the upper bound of its 95% interval as the value for $F(\mathbb{s}_{tar})$, to update the estimated T-value. Thus $ET(\mathbb{t})$ is iteratively updated by the following equation:

$$ET(\mathbb{t})^{'} = max\{Upper[F(\mathbb{s}_{tar})], \gamma \cdot \max_{\mathbb{t}_{suc} \in \mathbb{T}_{suc}} ET(\mathbb{t}_{suc})\} \tag{5}$$

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

Softmax Transition Selection. To cost-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 by assigning a selection probability to a transition proportional to the transition's T-value. The selection probability is given below (from [8]):

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

where $Prob(\mathbb{t}_{out}^{'})$ denotes the selection probability of an outgoing transition $\mathbb{t}_{out}^{'}$; $ET(\mathbb{t}_{out}^{'})$ 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 [10]. 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.

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

4.2 Uncertainty Generation Strategies

Since SH-CPSs typically operate in an uncontrolled environment [2] and are constantly affected by various environmental uncertainties, e.g., measurement uncertainties from sensors and actuation deviation from actuators. Effects of these uncertainties on SH-CPSs' behaviors should be explicitly considered and tested.

In our previous work [3], we adopted three levels of uncertainty from [11] and provided modeling notions to explicitly capture uncertainties that affect the behaviors of SH-CPSs. Table 1 presents a summary of the three uncertainty levels, with their definitions, methods for specification, and generation mechanisms.

For level 1 uncertainty, at a given point of time, the value of an uncertain factor is a single value with a margin of error, such as the precision of a digital compass. Based on its specification, its precision can be determined with a margin of error. The determined value and the margin of error are specified as an interval. By selecting a value from the interval, level 1 uncertainty can be simulated.

Level 2 uncertainty signifies the situation that an uncertain factor has alternative values with known probabilities, like the measurement error of an accelerometer. By statistically analyzing samples of the measurement error, the probability distribution of the measurement error can be obtained. Based on the distribution, a value can be generated for the uncertain factor to simulate level 2 uncertainty.

For level 3 uncertainty, an uncertain factor also has multiple possible values, while only ranked likelihoods rather than probabilities of the possible values are known. In this case, possibility distribution is used to capture the ranked likelihoods. For instance, wind speed and direction are level 3 uncertainties, since the probability of each possible value is unknown due to limited knowledge and we can only compare their likelihood. To simulate level 3 uncertainty, the possibility distribution is first transformed to an equivalent probability distribution [12], from which the value of the uncertain factor is generated.

Based on testers' domain knowledge, relevant environmental uncertainties can be explicitly modeled at the three levels (see [3] for further details). By simulating the uncertainties based on the specification, effects of uncertainties are reflected in the testing environment, which enables SH-CPSs to be tested under uncertainties.

Table 1 Uncertainty Level, Definition, Specification and Generation

Level	Definition	Specification	Generation
1	"A determined value with a margin of error" [13]	Interval	Derive an uncertainty value from the interval.
2	A set of possible values with known probability for each value [13]	Probability distribution	Generate an uncertainty value according to the probability distribution.
3	A set of possible values with known likelihood for each value [13]	Possibility distribution	Transform the possibility distribution to an equivalent probability distribution [12]. Based on it, generate an uncertainty value.

4.3 Implementation

We implemented the FOT algorithm and the three uncertainty generation strategies in TM-Executor. Fig. 2 presents its three packages: software in the loop testing (light gray), uncertainty generation (dark gray), and FOT (white).

TM-Executor tests the software of an SH-CPS 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' inputs and outputs to simulate the effects of uncertainties. Based on uncertainty specification, an uncertainty generator generates the values of uncertain factors whenever sensor data or actuation instructions are transferred between the software and simulators. By using the values to modify simulators' inputs and outputs, the specified uncertainties are introduced into the testing environment.

The SUT and its ETM are executed together by an execution engine, which is deployed in Moka [14], a UML model execution platform. During the execution, the engine dynamically derives a DFSM from the ETM and used it to guide the execution. Meanwhile, the active state's state invariant is checked by a test inspector (using DresdenOCL [7]). The inspector evaluates the invariant with the actual values of the state variables, which are updated by the execution engine via testing APIs (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 Equation (1). Taking fragility as input, the FOT algorithm updates its estimate of T-value (Equation (5)) and uses the softmax policy to select the next transition. Next, the test driver generates a valid test input with EsOCL [6], 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

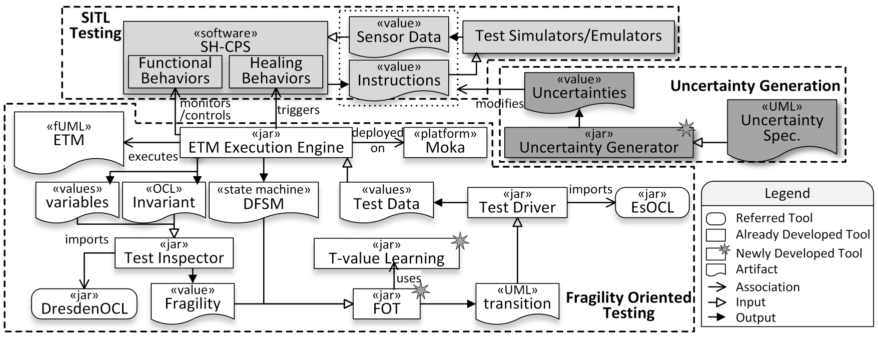

Fig. 2 SH-CPS Testing Framework

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 cost-effectively find faults.

5. Evaluation

We aim to evaluate the cost-effectiveness of FOT by comparing it with a Coverage Oriented Testing (COT) algorithm to test nine self-healing behaviors in three real SH-CPSs under 10 uncertainties, by answering two research questions: **RQ1:** Is FOT more effective than COT in terms of fault revelation? **RQ2:** Compared with COT, does FOT incur less cost to find a fault?

5.1 Case Studies and Test Configuration

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 and has five self-healing behaviors; 2) ArduRover[9] is an autopilot system for ground vehicles having two self-healing behaviors to avoid obstacle and handle the disruption of control link; 3) ArduPlane[10] 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, gyro meter, compass, and servo 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. More details can be found in [5]. The system specification includes 10 uncertainties related to the sensors and actuators and details are presented in Table 2.

Table 2 Identified Uncertainties from the Three Case Studies

Uncertainty	Level	Specification	Uncertainty	Level	Specification
Wind Direction	3	Possibility Categorical Distribution	Servos Bias	2	Probability Normal Distribution
Wind Velocity			Barometer Altitude Noise		
GPS Location Noise	2	Probability Normal Distribution	Barometer Climb Rate Noise		
GPS Velocity Noise			Accelerometer Noise		
GPS Location drift			Compass Noise		

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

5.2 Experimental Design and Execution

Table 3 is the experiment design. We implemented COT [4] and used it as the comparison baseline. It selects a transition with a likelihood that is reverse proportional to the total number of times that the transition has been fired plus one, to explore uncovered transitions as many as possible. For FOT, we set discount rate γ to 0.99 and temperature τ to 0.2.

Table 3 Experiment Design

RQ	Comparison	Case Study		#Runs	Metric	Statistical Test
		Name	#Self-Healing Behaviors			
1	FOT vs. COT	ArduCopter	5	10	*NDF*	Fisher's exact test, odds ratio
2		ArduPlane	2		*TFF*	Vargha and Delaney's $\widehat{A}_{12}$ Welch's t-test
		ArduRover	2			

The nine self-healing behaviors were tested independently (i.e., nine experiments). We specified expected functional and self-healing behaviors as ETMs, whose statistics are shown in Table 4. The last row of Table 4 presents twice the time taken by COT to cover all transitions of an ETM. We chose this time as the maximum test execution time for each ETM. To reduce randomness, we run each experiment 10 times for both algorithms.

Table 4 Descriptive Statistics of ETMs

Statistics	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
Max. Exe. Time (mins)	744	472	562	740	276	960	756	914	290

To answer RQ1, we used *Number of Detected Faults* (*NDF*) to evaluate the effectiveness of the algorithms in terms of finding faults and is calculated as $NDF = \sum_{i=1}^{10} n_i$, where n_i is the number of detected faults in the *i*th run of an experiment. Note that it is the first time the three case studies are tested under uncertainties. Thus, the total number of real faults was unknown. For RQ2, we define *Time to Find Fault* (*TFF*, i.e., the time (in minutes) that FOT/COT spent to find a fault) to assess the cost of finding faults. TFF_i represents the *TFF* for the *i*th run of an experiment and the average *TFF* of an experiment is $\overline{TFF} = \frac{\sum_{i=1}^{m} TFF_i}{m}$, where *m* is the number of runs out of ten, that a fault was detected.

Following the guidelines in [15], we conducted the Fisher's exact test to check the significance of results and used the odds ratio as the effect size measure for the results of RQ1, as they are dichotomous data, i.e., faults found or not. Since data to answer RQ2 are continuous, i.e., the time to find a fault, we applied the Vargha and Delaney's $\widehat{A}_{12}$ statistics to measure the effect size. To check the significance of the results, we first performed the Shapiro-Wilk test to test the normality of the two *TFF* samples. The calculated p-values corresponding to the two algorithms are 0.26 and 0.48, which are greater than 0.05 suggesting that the samples do not strongly depart from normality. Based on this, we performed the Welch's t-test to check the significance of RQ2's results, because the two *TFF* samples have unequal variances as results of the F-test revealed.

5.3 Evaluation Results

Table 5 shows results for RQ1. Within the fixed time, FOT and COT detected the same fault for SH1. FOT was able to find faults in five other self-healing behaviors, while COT failed for the rest. Note that both FOT and COT achieved 100% transition coverage and thus we do not compare them based on this measure.

For SH1, the Fisher's exact test calculated a p-value of 0.474 (greater than 0.05) suggesting no significant difference between COT and FOT. The obtained odds ratio was 3.67 indicating that FOT is likely to find more faults than COT. Both FOT and COT didn't detect any fault in SH4, SH5, and SH6, which might be due to two reasons: 1) No faults in these behaviors, 2) Neither algorithm covered a particular path with specific test input and uncertainty values that could reveal faults. For the other five behaviors, COT failed to detect any faults, while FOT succeeded in all the cases suggesting that FOT is significantly better than COT.

Since COT only detected a fault in SH1, only the *TFFs* for SH1 are used to answer RQ2. On average, FOT could find a fault within 142 minutes, while COT required 282 minutes to find a fault (Table 5). We conducted the Welch's t-test and the result (p-value = 0.043) showed that COT took significantly more time than FOT to find a fault. In addition, the result of $\hat{A}_{12} = 0.875$ suggests that, in most cases, COT is expected to spend more time than FOT to find a fault.

Table 5 Number of Detected Faults in 10 Runs

Alg.	Metrics	ArduCopter					ArduRover		ArduPlane	
		SH1	SH2	SH3	SH4	SH5	SH6	SH7	SH8	SH9
COT	*NDF*	8	0	0	0	0	0	0	0	0
	$\overline{TFF}$ (mins)	282	-	-	-	-	-	-	-	-
FOT	*NDF*	10	7	6	0	0	0	8	10	5
	$\overline{TFF}$ (mins)	142	247	415	-	-	-	548	468	208

5.4 Discussion

We obtained 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 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 generate uncertainties according to interval, probability or possibility distributions. Though this may not be the optimal strategy, it provides a preliminary solution for this problem. Second, a typical objective of a coverage-based testing approach is to achieve full coverage, e.g., 100% transition coverage. However, this is not sufficient to reveal a fault in self-healing behaviors under uncertainty, as demonstrated by the experiment result. Since transitions have different possibilities to reveal a fault, the ones with high possibility should be tested more frequently to cost-effectively find faults. Third, for testing SH-CPSs under uncertainty, FOT is more cost-effective than COT in terms of a number of detected faults and time spent to reveal a fault. On average, FOT found 80% more faults and when both algorithms managed to find a fault, FOT took 50% less time than COT. This is because FOT used execution information to dynamically learn transitions' T-values, which indicates the possibility of revealing a fault when firing transitions; COT only used coverage information to direct test execution.

5.5 Threats to Validity

Conclusion validity is concerned with factors that affect the conclusion drawn from the outcome of experiments. Because of random transition selection used by FOT and COT, randomness in the results is the most probable conclusion validity threat. To reduce this threat, all the experiments were repeated 10 times. We applied two statistical tests and two effect size measures to evaluate statistical differences and magnitude of improvement. In addition, the variation in simulated uncertainties may be another conclusion validity threat. However, we simulated the same sequence of uncertainties for both algorithms. We fixed the maximum test execution time for both of the algorithms. This measure was taken to remove the **internal validity threat** that different settings might favor one algorithm over the other. However, more experiments with other settings in terms of test execution time are required to further strengthen the current conclusion. **External validity threats** concern the generalization of the experiment results. We tested nine self-healing behaviors of three real case studies. However, additional case studies are needed to further generalize the results. With respect to **construct validity threats**, we used the number of detected faults and the time required to detect a fault as the evaluation metrics, which are comparable across both of the algorithms.

6. Related Work

Model-Based Testing (MBT) has shown good results of producing effective test suites to reveal faults [16]. For a typical MBT approach, abstract test cases are generated from models first, e.g., using structural coverage criteria (e.g., all state coverage) [17, 18]. 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 [19, 20]. De Vries et al. [19] 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. [20] 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, FOT relies on the model execution of ETMs to facilitate the testing of SH-CPSs under uncertainty. During the execution, FOT applies a reinforcement learning technique to learn transitions' T-values, which direct FOT to cost-effectively find faults. Besides, FOT focuses on testing self-healing behaviors in the presence of environmental uncertainty, which is not covered by existing works. The first reinforcement learning based testing algorithm was proposed in [4]. It uses frequencies of transitions' coverage as the heuristics of reinforcement learning. By learning frequencies, the algorithm tries to equally explore all transitions. However, a long-term reward is not realized in this approach. Groce et al. [21] created a framework to simplify the application of reinforcement learning for testing, which uses coverage as the heuristic and relies on SARSA(λ) [8] for calculating long term rewards. Similarly, Araiza-Illan et al [22] 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.

7. Conclusion

This paper presents a new testing algorithm, Fragility Oriented Testing (FOT), for testing self-healing behaviors of SH-CPSs under uncertainty. It applies model execution and a reinforcement learning method to learn each transition's T-value, which indicates the possibility to reveal a fault after firing the transition. Accordingly, FOT focuses on exercising transitions with high T-values to cost-effectively find faults. To evaluate FOT, we tested nine self-healing behaviors in three case studies. The results showed that FOT significantly outperformed COT for five out of nine self-healing behaviors in terms of faults finding. On average, FOT discovered 80% more faults than COT. When both algorithms succeeded to find a fault, FOT on average took 50% less time than COT. In the future, we plan to conduct more experiments and integrate more advanced reinforcement learning algorithms to further enhance the algorithm's fault detection capability.

References

1. T. Bures, D. Weyns, C. Berger *et al.*: Software Engineering for Smart Cyber-Physical Systems--Towards a Research Agenda: Report on the First International Workshop on Software Engineering for Smart CPS. In: ACM SIGSOFT Software Engineering Notes, vol. 40, no. 6, pp. 28-32. ACM (2015)
2. E. A. Lee: Cyber physical systems: Design challenges. In: 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC) pp. 363-369. IEEE (2008)
3. T. Ma, S. Ali, and T. Yue: Modeling Healing Behaviors of Cyber-Physical Systems with Uncertainty to Support Automated Testing. In: Simula Research Lab Technical Report 2016-08 (2016). https://www.simula.no/publications/modeling-healing-behaviors-cyber-physical-systems-uncertainty-support-automated-testing
4. M. Veanes, P. Roy, and C. Campbell: Online testing with reinforcement learning. In: Formal Approaches to Software Testing and Runtime Verification, pp. 240-253. Springer (2006)
5. T. Ma, S. Ali, and T. Yue: Fragility-Oriented Testing with Model Execution and Reinforcement Learning. In: Simula Research Lab Technical Report 2017-05 (2017). https://www.simula.no/publications/fragility-oriented-testing-model-execution-and-reinforcement-learning

6. S. Ali, M. Z. Iqbal, A. Arcuri *et al.*: Generating test data from OCL constraints with search techniques. In: IEEE Transactions on Software Engineering, vol. 39, no. 10, pp. 1376-1402. IEEE (2013)
7. B. Demuth, and C. Wilke: 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 (2009)
8. R. S. Sutton, and A. G. Barto: Reinforcement learning: An introduction. MIT press Cambridge (1998)
9. C. Z. Mooney, R. D. Duval, and R. Duval: Bootstrapping: A nonparametric approach to statistical inference. Sage (1993)
10. Y. Anzai: Pattern recognition and machine learning. Elsevier (2012)
11. W. E. Walker, R. J. Lempert, and J. H. Kwakkel, Deep uncertainty, Encyclopedia of operations research and management science, pp. 395-402. Springer (2013)
12. D. Dubois, H. Prade, and S. Sandri, On possibility/probability transformations, Fuzzy logic, pp. 103-112. Springer (1993)
13. T. Ma, S. Ali, and T. Yue: Conceptually Understanding Uncertainty in Self-Healing Cyber-Physical Systems. In: Simula Research Lab Technical Report 2016-07 (2016). https://www.simula.no/publications/conceptually-understanding-uncertainty-self-healing-cyber-physical-systems
14. J. Tatibouet: Moka – A simulation platform for Papyrus based on OMG specifications for executable UML. In: EclipseCon. OSGI (2016)
15. A. Arcuri, and L. Briand: 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 (2011)
16. E. P. Enoiu, A. Cauevic, D. Sundmark *et al.*: A Controlled Experiment in Testing of Safety-Critical Embedded Software. In: IEEE International Conference on Software Testing, Verification and Validation (ICST) pp. 1-11. IEEE (2016)
17. M. Utting, A. Pretschner, and B. Legeard: A taxonomy of model-based testing approaches. In: Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297-312. Wiley (2012)
18. W. Grieskamp, R. M. Hierons, and A. Pretschner: Model-Based Testing in Practice. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2011)
19. R. G. de Vries, and J. Tretmans: On-the-fly conformance testing using SPIN. In: International Journal on Software Tools for Technology Transfer (STTT), vol. 2, no. 4, pp. 382-393. Springer (2000)
20. K. G. Larsen, M. Mikucionis, and B. Nielsen: Online testing of real-time systems using uppaal. In: International Workshop on Formal Approaches to Software Testing pp. 79-94. Springer (2004)
21. A. Groce, A. Fern, J. Pinto *et* al.: Lightweight automated testing with adaptation-based programming. In: IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE) pp. 161-170. IEEE (2012)
22. D. Araiza-Illan, A. G. Pipe, and K. Eder: 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 (2016)