UiO : University of Oslo
Faculty of mathematics and natural sciences

# Executable Model Based Testing for Self-Healing Cyber-Physical Systems Under Uncertainty

Tao Ma

Thesis submitted for the degree of Ph.D.

Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
December 2020

# Abstract

Self-healing is becoming a critical feature of Cyber-Physical Systems (CPSs). By detecting faults and applying recovery adaptations at runtime, self-healing behaviors can help CPSs to maintain functional normal in the presence of faults. CPSs with the self-healing feature are named as Self-Healing CPSs (SH-CPSs). Besides recovery, SH-CPSs have to deal with various uncertainties, such as measurement errors from sensors and actuation deviations from actuators. To assess the dependability of SH-CPSs, it is necessary to test if SH-CPSs can still behave as expected under uncertainty. However, the autonomy of self-healing behaviors and the impact of uncertainties make it challenging to conduct such testing. To this end, an executable model-based testing approach is proposed in this thesis. In this approach, the expected behaviors of the SH-CPS under test are specified as an executable test model. By executing the SH-CPS together with the test model, sending them the same test inputs, and comparing their consequent states, we can dynamically test the system against its test model.

To realize this executable model-based testing approach, five contributions have been made and are presented in this thesis: (**C1**) a Conceptual Model of SH-CPS and Uncertainty (CMSU), for constructing a comprehensive and precise understanding of CPS, self-healing and associated uncertainty; (**C2**) a Modeling framework of SH-CPS (MoSH), to facilitate the creation of an executable test model that captures the expected behaviors of the SH-CPS under test; (**C3**) a testing framework (TM-Executor), for testing an SH-CPS against a test model via co-execution of the system and the model; (**C4**) a Fragility-Oriented Testing (FOT) approach, to learn the optimal policies of choosing test inputs for fault detection; (**C5**) an empirical study, to find the best reinforcement learning algorithms for detecting faults in the SH-CPS under uncertainty.

By applying MoSH to create executable test models and employing TM-Executor and FOT to test diverse SH-CPSs, we demonstrate that it is practical to apply the executable model-based approach to test SH-CPSs under uncertainty. The fault detection ability of the fragility-oriented testing approach is significantly higher than random testing and

coverage-oriented testing. Reinforcement learning algorithms have shown competence in detecting faults in SH-CPSs under uncertainty. Based on results of the empirical study, we found that the combination of Q-learning and Uncertainty Policy Optimization algorithms managed to detect the most faults in selected six SH-CPSs. On average, they managed to discover two times more faults than the other reinforcement learning algorithms.

# Acknowledgements

Five years ago, I decided to quit my job and became a Ph.D student in software engineering department at Simula. Although it was a bit challenging at the beginning, as a completely fresh guy in the software engineering field, the decision has offered me a great opportunity to expand my boundary and make me more clearly understand myself. Definitely, doing a Ph.D. is not an easy journey. I am really grateful to all the people around me.

First and foremost, I would like to thank my supervisors Shaukat Ali and Tao Yue. Thanks for your trust that allows me to start my Ph.D. journey. Thanks for your patient listening, insightful comments, and valuable teaching and discussions that help me to know how to solve a research problem rigorously and systematically.

To all my colleagues, I must express my gratitude for all your supports. At the beginning of my Ph.D., I only know few about the research works in software testing and model-based engineering. The valuable discussions with you helped me to much more quickly get familiar with these new fields. Without you, I cannot imagine how I could conquer the challenges I have met through the journey.

To my beloved wife, Hong, thanks for your accompanies, support, inspiration, and sacrifice. You are the most precious gift I have gotten in my life. Also, thanks to all my family. Thanks for your care and support.

# List of Papers

My entire PhD work leads to five pulications (three journal papers and two conference papers). Since Paper B is a journal extension of Paper D and Paper A is presented as a journal first paper (Paper E), these two papers (D and E) are not included in the thesis to avoid redundancy.

**Paper A. Modeling Foundations for Executable Model-Based Testing of Self-Healing Cyber-Physical Systems**

T. Ma, S. Ali, and T. Yue.

Journal of Software & Systems Modeling (SOSYM). DOI: 10.1007/s10270-018-00703-y

**Paper B. Testing Self-Healing Cyber-Physical Systems under Uncertainty: A Fragility-Oriented Approach**

T. Ma, S. Ali, T. Yue, and M. Elaasar.

Software Quality Journal (SQJ). DOI: 10.1007/s11219-018-9437-3

**Paper C. Testing Self-Healing Cyber-Physical Systems under Uncertainty with Reinforcement Learning: An Empirical Study**

T. Ma, S. Ali, and T. Yue.

Journal of Empirical Software Engineering (EMSE). DOI: 10.1007/s10664-021-09941-z

**Paper D. Fragility-Oriented Testing with Model Execution and Reinforcement Learning**

T. Ma, S. Ali, T. Yue, and M. Elaasar.

In: IFIP International Conference on Testing Software and Systems (ICTSS 2017). DOI: 10.1007/978-3-319-67549-7_1

**Paper E. Modeling Foundations for Executable Model-Based Testing of Self-Healing Cyber-Physical Systems**

T. Ma, S. Ali, and T. Yue.

Journal first paper presented in: IEEE International Conference on Software Testing, Verification and Validation (ICST 2020)

# Contents

# Part I

Summary

# Summary

# 1 Introduction

Cyber-Physical Systems (CPSs) are integrations of computation and physical processes [1]. In CPSs, computing components monitor and control physical processes via sensors and actuators, and cooperate with each other via network communication. The integration of computation, communication, and control awards CPSs a higher level of intelligence, which enables them to adapt and optimize their behaviors autonomously at runtime [2]. One of such autonomous features is self-healing, which endows CPSs with the ability to detect and recover from errors caused by software or hardware faults at runtime. CPSs with the self-healing feature are referred as Self-Healing CPSs (SH-CPSs).

Besides recovery, SH-CPSs have to deal with various uncertainties arising from measurements acquired by sensors and actuations conducted by actuators. In reality, the exact value of a measurement error or an actuation deviation is unknown, and the uncertain value will affect the behaviors of SH-CPSs and may prohibit the systems from behaving as expected. To assess the dependability of SH-CPSs, it is necessary to test if an SH-CPS can still behave as expected under uncertainty.

To tackle this testing problem, we proposed an executable model-based testing approach. In this approach, a test model is used to capture components and expected behaviors of the SH-CPS under test. To test the system, the test model is executed together with the SH-CPS and simulators of sensors, actuators, and environment. By introducing uncertainties via the simulators, sending the same test inputs to the system and the test model, and comparing their consequent states, we can therefore determine if the system behaves as

expected, and also evaluate how likely the system is going to behave differently from the model. The likelihood is defined as *fragility*. It is used as a heuristic to effectively find test inputs leading to an unexpected behavior. To realize such an executable model-based testing approach, a series of model-based methodologies was proposed with respect to five contributions, as shown in Figure 1.

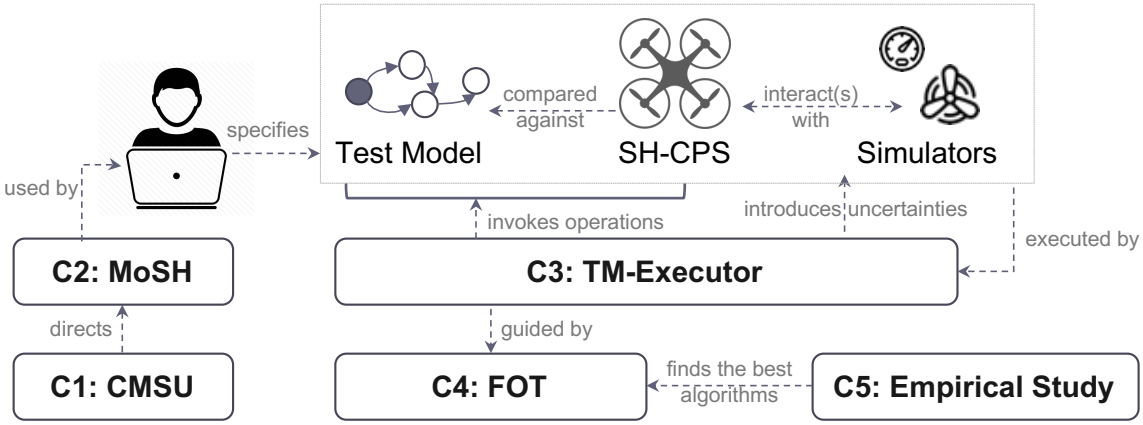


Figure 1 Scope of Executable Model-Based Testing for Self-Healing Cyber-Physical Systems Under Uncertainty

First of all, a Conceptual Model of SH-CPS and Uncertainty (C1: CMSU in Figure 1) was proposed [3]. CMSU captures key concepts and their relationships about CPS, functional and self-healing behaviors, and uncertainty. The conceptual model forms a common understanding of what an SH-CPS is and what the uncertainty means for it.

Based on CMSU, a Modeling framework Of SH-CPS (C2: MoSH in Figure 1) was developed to facilitate the specification of an executable Test Model (TM) that captures the expected behaviors and involved uncertainties of the SH-CPS under test [3]. A TM uses UML[1] class diagrams to capture system components and UML state machines to specify expected behaviors of each component. To enable the TM execution, we restrict the modeling notations to the executable subset of UML, which is defined by the standards of Semantics of a Foundational Subset for Executable UML Models (fUML) [4] and Precise Semantics of UML State Machines (PSSM). Several metaclasses in the subset were extended by the stereotypes defined in MoSH to specify self-healing behaviors and uncertainties. MoSH also provides a modeling methodology to guide how to use UML class diagrams and state machines along with MoSH to create an executable TM.

[1] Unified Modeling Language (UML): https://www.omg.org/spec/UML/2.5.1/PDF

To execute a TM together with the SH-CPS under test, we implemented a testing framework (C3: TM-Executor in Figure 1), based on standards of fUML [4], PSSM [5], Functional Mockup Interface (FMI) [6], and our extensions to fUML and PSSM [3]. fUML, PSSM and the extensions provide execution semantics of model elements in a TM and the semantics allow it to be executed in a deterministic manner. The FMI standard provides standard interfaces to enable the co-execution of hybrid models, such as the TM and simulation models that are constructed with diverse modeling paradigms. Based on the FMI standard, we devised a co-execution algorithm [3]. By applying the algorithm, TM-Executor orchestrates the execution of the SH-CPS, TM and simulators, and allows us to dynamically test an SH-CPS against a TM under a set of identified uncertainties.

To effectively detect faults in SH-CPS under uncertainty, a Fragility Oriented Testing (C4: FOT in Figure 1) method was proposed [7]. In FOT, we evaluated the likelihood that the system under test is going to behave inconsistently with its TM. The likelihood is defined as *fragility*. We devised two reinforcement learning based algorithms that take the fragility as a reward to find optimal policies of invoking operations (i.e., interfaces of the system) and introducing uncertainties, respectively. More specifically, the two algorithms take the state of the system as input, and output an operation invocation and a value for each uncertainty. After performing the invocation on the system or introducing the uncertainties, the algorithms obtain the fragility of the system, provided by TM-Executor, and use the fragility to learn how to choose the invocation and uncertainty values to reach the highest fragility and detect faults.

Afterward, we conducted an empirical study (C5 in Figure 1) to find the best reinforcement learning algorithms for the FOT approach. In this study, we evaluated the performance of 14 combinations of reinforcement learning algorithms, on six SH-CPSs. Results of the empirical study reveal that Q-learning [8] and Uncertainty Policy Optimization [7] managed to detect the most faults. On average, this combination found two times more faults and took 52% less time to find a fault than the others.

In summary, this thesis provides a complete model-based solution for testing SH-CPSs under uncertainty. With the UML extensions and modeling methodology provided by MoSH, testers can systematically specify a test model to capture components, expected behaviors, and uncertainties of the SH-CPS under test. TM-Executor enables the test model to be executed together with the SH-CPS in a simulated environment, and a Fragility

Oriented Testing (FOT) method and two reinforcement learning based algorithms were devised to effectively detect faults in the SH-CPS under uncertainty.

The thesis is composed of two parts. The first part (Part I) summarizes all research works covered by the thesis. Section 2 provides background information required to understand the thesis. Section 3 presents the research methodology, followed by the contributions of the thesis and key results in Section 4 and Section 5 respectively. Section 6 discusses possible future directions and Section 7 concludes the thesis. The second part (Part II) presents the related papers with respect to the five contributions. Figure 2 shows the mapping between the summary and the collection of papers.

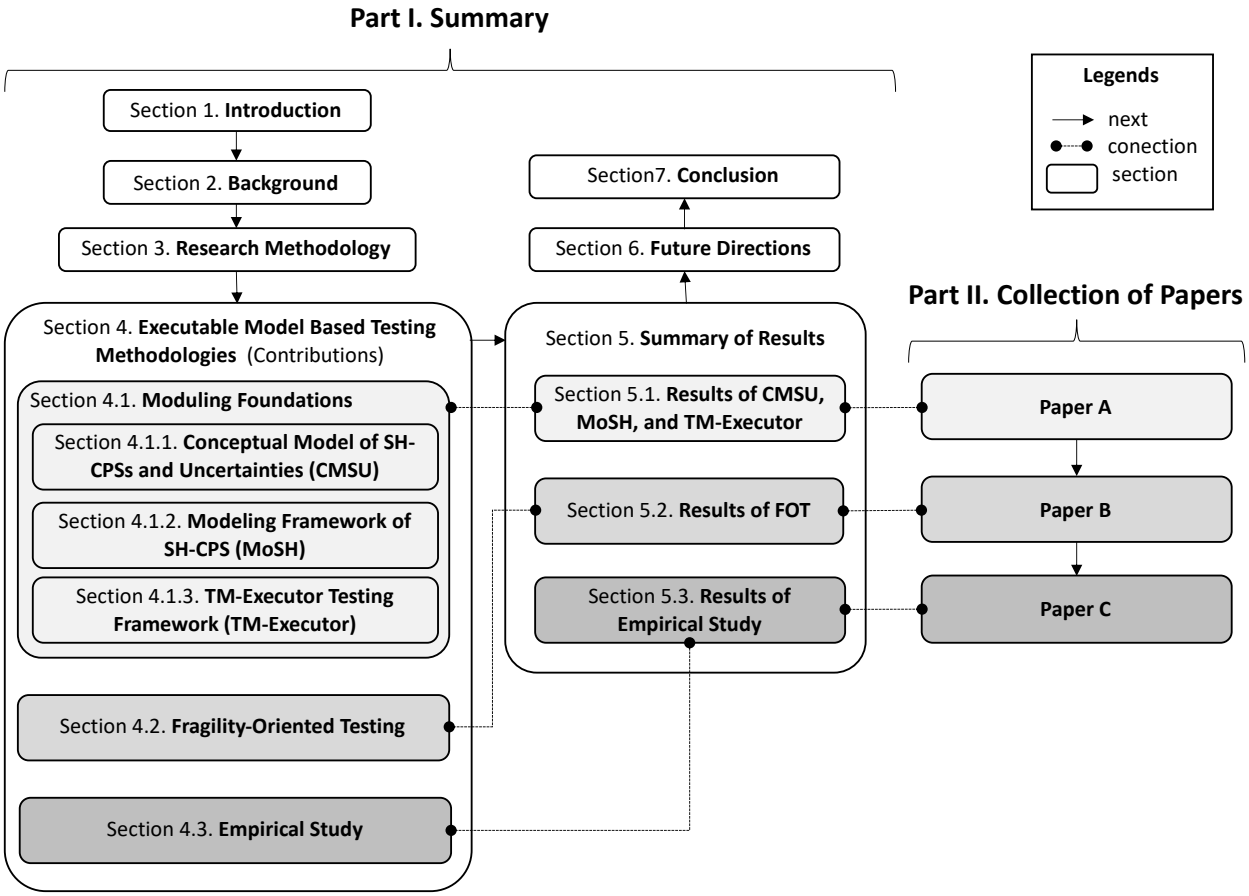


Figure 2 Thesis Structure

# 2 Background

This section briefly introduces SH-CPS and uncertainty in Section 2.1. Section 2.2 explains several standards that build the foundation for executing the SH-CPS under test together with its test model. An overview and limitations of existing model-based testing approaches are given in Section 2.3, and Section 2.4 introduces the general idea of reinforcement learning.

## 2.1 Self-healing Cyber-Physical System and Uncertainty

An SH-CPS can be seen as a set of heterogeneous and distributed physical units. Each physical unit has one or more controllers that use sensors and actuators to monitor and control some physical processes, and the controllers are cooperating with each other via network communication. Moreover, the SH-CPS is equipped with probes and effectors that allow the system to monitor its internal state and to make runtime adaptations. Based on the measurement generated by probes, the controllers employ fault detection algorithms to determine if a software or hardware fault has occurred, and apply recovery policies to adapt the system's parameters, components, or behaviors via effectors, to recover the system from the detected fault. The fault detection algorithms and recovery policies are the self-healing behaviors of an SH-CPS.

As SH-CPSs typically operate in an uncontrolled environment, the behaviors of SH-CPSs are affected by various uncertainties. For instance, the measurement from sensors and the actuation performed by actuators are affected by uncertain measurement errors and actuation deviations. These uncertainties may prevent them from behaving as expected. This thesis provides a model-based solution for testing if an SH-CPS can still behave as expected, under a set of identified uncertainties.

## 2.2 Executable Models and Hybrid Co-execution

Model-based system engineering has emerged as a standard approach for engineering complex, interdisciplinary systems [9]. Model is the key to this approach. It provides a systematic and precise way to capture and specify the system under development. To avoid ambiguity and facilitate validation, the model is becoming executable, with execution semantics of the model defined in standards. One example is the Unified Modeling

Language (UML). The standards of fUML [4] and PSSM [5] define the semantics of a subset of UML elements. Such semantics enable UML activity diagrams and state machines to be executed in a deterministic manner.

For complex, interdisciplinary systems, such as CPSs, it typically involves multiple models created under diverse modeling paradigms. For example, the physical process of a CPS could be represented as a continuous-time model, such as differential equations, while computation logics are described as state machines or activity diagrams. As a continuous-time model and a discrete model have different time semantics, a co-execution framework is needed to orchestrate the executions of the hybrid models. Toward this direction, the FMI (Functional Mockup Interface) standard [6] defines the interfaces that enable data exchange among different models. A co-execution algorithm needs to be provided to determine the order and interval of the data exchanges. In this thesis (Paper A), we have devised such an algorithm to co-execute a test model with the SH-CPS under test.

## 2.3 Model-Based Testing

Model-Based Testing (MBT) is a sub-field of model-based system engineering. It uses a test model to capture the expected behaviors of the system under test and/or the behaviors of its environment. Pairs of input and output of the model can be automatically derived from the model and used as test cases to test the system, i.e., the output of the model is the expected output of the system.

In a traditional MBT approach, all test cases are generated from a test model, guided by test selection criteria, such as covering all states and transitions. Afterward, the test cases are run on the system under test, and test verdicts are generated by comparing the output of the system and the expected one contained in the test case.

A challenge to apply this traditional MBT approach to test SH-CPS is how to choose the proper test selection criteria. Aggressive criteria, such as covering all possible transition sequences, will lead to too many or even infinite test cases that are impractical to run all of them. Although fewer test cases can be obtained by conservative criteria, the fault detection ability of the smaller test suite is limited.

To overcome this dilemma, an executable model-based testing approach is proposed in this thesis. In this approach, the system under test is dynamically tested against a test model. By learning the results of performed test cases, an intelligent testing algorithm can

identify the fragile parts of the system. By focusing on testing the fragile parts, the algorithm is expected to reveal faults more effectively.

## 2.4 Reinforcement Learning

To detect faults in SH-CPS under uncertainty, we need to find the optimal policy of choosing test inputs. Finding such an optimal policy is exactly the goal of reinforcement learning. Via a trial-and-error approach, reinforcement learning algorithms learn the long-term rewards of candidate actions. By choosing the action with the highest reward, they can find the best policy of choosing actions and obtain the highest reward.

In the context of testing SH-CPS under uncertainty, the candidate actions are operation invocations that can be performed on the system to control its behaviors, and uncertainty values that are to be introduced via simulators to mimic the effect of uncertainties. The reward of choosing an action is the likelihood that an unexpected behavior can be detected after conducting the action. We define this likelihood as fragility. Its value is between zero and one. When fragility equals one, it means an unexpected behavior is detected. Taking the fragility as a reward, we have devised two reinforcement learning based algorithms to find the optimal policies of choosing operation invocations and uncertainty values respectively. Besides, we conducted an empirical study to compare the performance of state-of-the-art reinforcement algorithms on testing SH-CPSs under uncertainty.

# 3 Research Methodology

This section presents the research methodology, including problem identification in Section 3.1, problem formulation in Section 3.2, solution realization and implementations in Section 3.3, and evaluation methods in Section 3.4.

## 3.1 Problem Identification

As SH-CPSs typically operate in an uncontrolled environment, they are affected by various uncertainties that may prohibit them from behaving as expected. To assess their dependability, it is necessary to test SH-CPSs under uncertainty. Based on the literature [1, 2, 10-12], we identified following challenges to solve this testing problem.

**Challenge 1.** *Self-healing and uncertainty are not precisely defined*. Due to lack of a rigorous definition, the term self-healing is used mixed with self-adaptive, self-* properties, fault tolerant and system dependability [13, 14]. However, the actual meaning of self-healing is the ability to detect faults and apply proper adaptations to recover from the faults, which are different from fault tolerance and the other self-* properties. The key components of fault detection and recovery have to be identified prior to testing self-healing systems. Regarding uncertainty, it can originate from numerous resources, such as missing or ambiguous requirements, inadequate design due to incomplete knowledge, and unpredictable environment [15, 16]. In this thesis, we focus on testing SH-CPSs under uncertainty arising from the interaction between the systems and their environment. We need to identify the uncertainties that impact the behaviors of SH-CPSs and explicitly specify the uncertainties to enable the uncertainty-aware testing.

**Challenge 2.** *Lacking a modeling methodology to enable model-based testing of SH-CPSs under uncertainty*. Model-based testing provides a systematic and automated approach for testing complex systems [17]. A test model is the main artifact in this approach. It needs a systematic modeling methodology to develop such a test model that can capture the expected behaviors and uncertainties of the SH-CPS under test and solve our testing problem.

**Challenge 3.** *Difficult to generate test cases offline due to self-adaptability*. The self-healing behaviors of SH-CPSs enable the systems to autonomously adapt their behaviors at runtime to recover from detected faults. As the expected behaviors could be adapted

dynamically, for a given input, the expected output depends on if one or more faults has occurred and if self-healing behaviors have detected them. Consequently, it will be challenging to generate test cases, i.e., pairs of input and expected output, for the complex behaviors of SH-CPSs.

**Challenge 4.** *Lacking a testing framework to test SH-CPSs under uncertainty*. As it could be too expensive or unsafe to test SH-CPSs in the real world, it is preferable to test them in a simulated environment. Uncertainties, like measurement errors from sensors, need to be introduced into the environment to simulate the effect of uncertainties and test if an SH-CPS can still behave as expected under the uncertainties. Consequently, it needs a testing framework to introduce the uncertainties and test SH-CPSs in an uncertainty-introduced environment.

**Challenge 5**. *Challenging to find the optimal testing policies*. Two types of policies are required to test SH-CPSs under a set of identified uncertainties. One is the policy for selecting operation invocations that control the behavior of the SH-CPS under test. The other is the policy used to choose the value of each identified uncertainty. The uncertainty values instantiate a concrete, uncertainty-introduced environment condition, in which the behavior of SH-CPS is to be tested. As the set of candidate operation invocations and uncertainty values is huge, an effective algorithm is needed to find these optimal policies to detect faults in the most effective manner.

## 3.2 Problem Formulation

After identifying the challenges, we formulate the following five research questions to address each of them.

**RQ1. What are self-healing and uncertainty?** Key concepts about these terms and the relations among the concepts need to be precisely defined to form a common understanding of SH-CPS and uncertainty. The identified concepts could also help us to capture the key components of SH-CPSs that have to be considered for testing.

**RQ2. How to specify expected behaviors and involved uncertainties of SH-CPSs?** To test if an SH-CPS can behave as expected under a set of identified uncertainties, we need to specify the expected behaviors and uncertainties first. As CPSs are hybrid and distributed systems, their expected behaviors are composed of the behaviors of their hybrid components. A systematic modeling methodology is required to capture the components

and their behaviors. Uncertainties that may affect any of the components also need to be specified to enable the uncertainty-aware testing.

**RQ3. How to test an SH-CPS against its expected behaviors?** After capturing the expected behaviors, we need to test if the SH-CPS can behave consistently with the specified expected behaviors. As it is difficult to pre-generate test cases before test execution (Challenge 3), ideally, an SH-CPS could be directly tested against its test model. To do so, it needs a testing framework that can execute an SH-CPS together with its test model and compare their behaviors during execution.

**RQ4. How to introduce uncertainties in a simulated environment?** In this thesis, we focus on the uncertainties arising from the interactions between SH-CPSs and their environment, like measurement errors and actuation deviations. To test if an SH-CPS can behave as expected under the uncertainties, effects of the uncertainties have to be reflected on the measurements from sensors and actions performed by actuators. Although specialized simulation models could be developed to simulate the effects of uncertainties, it will be cumbersome and error-prone to manually build the simulation model for each sensor and actuator. To facilitate testing SH-CPSs under the uncertainties, it needs a testing framework that can automatically introduce uncertainties during test executions, based on the specifications of the uncertainties.

**RQ5. How to find the optimal policies of selecting operation invocations and the values of uncertainties to effectively detect faults?** To detect a fault in an SH-CPS under uncertainties, it needs to find a sequence of operation invocations and a sequence of uncertainty values for each uncertainty that can work together to reveal the fault. As the numbers of candidate operation invocations and possible combinations of uncertainty values are huge, it is infeasible to cover all of them. It needs a novel approach that can learn how to choose the operation invocations and uncertainty values to increase the chance of detecting faults, and find faults in the most effective manner.

## 3.3 Solution Realization and Implementations

A conceptual model, a modeling framework, a testing framework, and a fragility-oriented testing approach were proposed to address the five research questions. Table 1 gives an overview of the solutions and implements.

Table 1 Solutions and Implementations

| RQ | Solution | Techniques/tools/languages | Implementations |
|---|---|---|---|
| 1 | *CMSU* | UML, OCL, Eclipse Papyrus | CMSU was implemented as UML class diagram with OCL constraints. Details of the conceptual model can be found in Paper A. |
| 2 | *MoSH* | UML, OCL, MARTE, Eclipse Papyrus | MoSH was implemented as four UML profiles with dependency on MARTE model library. Detailed specification and modeling methodology can be found in Paper A. |
| 3, 4 | *TM-Executor* | fUML, PSSM, FMI, Eclipse Papyrus Moka | TM-Executor was implemented as a plugin of Eclipse Papyrus based on Moka, a model execution engine for UML models. Details about the framework are introduced in Paper A. |
| 5 | *FOT* | Reinforcement learning, Eclipse Papyrus, JAVA, Python, Tensorflow | FOT includes two reinforcement learning based testing algorithms that are used to select operation invocations and uncertainty values respectively. They were implemented in JAVA and Python, and the algorithms can be used by TM-Executor to find the operation invocations and uncertainty values leading to unexpected behaviors. Details of this approach is explained in Paper B. |

For RQ1, a conceptual model of SH-CPS and uncertainty (CMSU) was derived from the literature. It captures key concepts about CPS, self-healing and uncertainty, as well as the relations among the concepts. The definition of each concept is also provided along with the model. These form the foundation of this thesis.

To facilitate the specification of expected behaviors of SH-CPSs and uncertainties (RQ2), a modeling framework (MoSH) was proposed based on the conceptual model. The framework provides four UML profiles and a modeling methodology to specify the expected behaviors and uncertainties of SH-CPSs, using UML class diagrams and state machines.

For RQ3 and RQ4, we proposed to test an SH-CPS by executing the system together with its test model. Based on extended standards of fUML [4], PSSM [5], and FMI [6], a testing framework (TM-Executor) was developed to enable the co-execution. Via simulators of sensors and actuators, TM-Executor can also introduce uncertainties during execution and enable the SH-CPS to be tested under uncertainties.

A fragility-oriented testing approach was devised to effectively find the operation invocations and uncertainty values that can lead to unexpected behaviors of the SH-CPS under test (RQ5). The approach includes two reinforcement learning based testing algorithms to select the operation invocations and uncertainty values respectively. By learning and focusing on operation invocations and uncertainty values that make an SH-

CPS more likely to behave differently from its test model, the algorithms can help TM-Executor to effectively reveal faults in the SH-CPS under a set of identified uncertainties.

## 3.4 Solution Evaluation

We selected several representative SH-CPSs from real world projects and the literature as case studies to empirically evaluate the applicability and performance of our proposed frameworks and testing approach, including Radio-Frequency Identification Supply Chain [18], Intelligent Service Robot [19], and ArduCopter [20], an autopilot system for aerial vehicles. For the modeling and testing frameworks, we applied the frameworks to create test models and test the selected SH-CPSs against the test models. By evaluating the modeling effort and measuring the cost of applying the testing framework to perform the executable model-based testing, we assessed if it is feasible to use the frameworks to test SH-CPSs under uncertainties. Regarding the testing approach, we compared its performance with the ones of state-of-the-art approaches, using broadly used evaluation metrics, such as state and transition coverage [21], number of detected faults [22], and time/space cost [23]. To reduce the effect of randomness and external factors, like the amount of time an approach can take to test an SH-CPS, we chose several different settings of the factors and conducted each testing task under each setting multiple times. Afterward, we performed statistical tests, such as Kruskal-Wallis test [24] and the Dunn's test [25] in conjunction with the Benjamini-Hochberg correction [26] and Vargha and Delaney statistics [27] to determine statistical significance and measure effect size. At last, we analyzed threats to validity of the evaluations and presented the measures we had taken to reduce these threats.

# 4 Executable Model Based Testing Methodologies

To test SH-CPSs under uncertainty, a series of executable model-based testing methodologies is presented in this thesis, including a modeling framework (MoSH) to create executable test models, a testing framework (TM-Executor) to co-execute the test model with the SH-CPS under test, and a fragility-oriented testing methodology (FOT) to effectively detect faults. Figure 3 presents an overview of the series of methodologies. An empirical study was also conducted to evaluate the performance of different reinforcement learning algorithms that can be used by FOT to find faults. The following subsections give a more detailed description of each work.

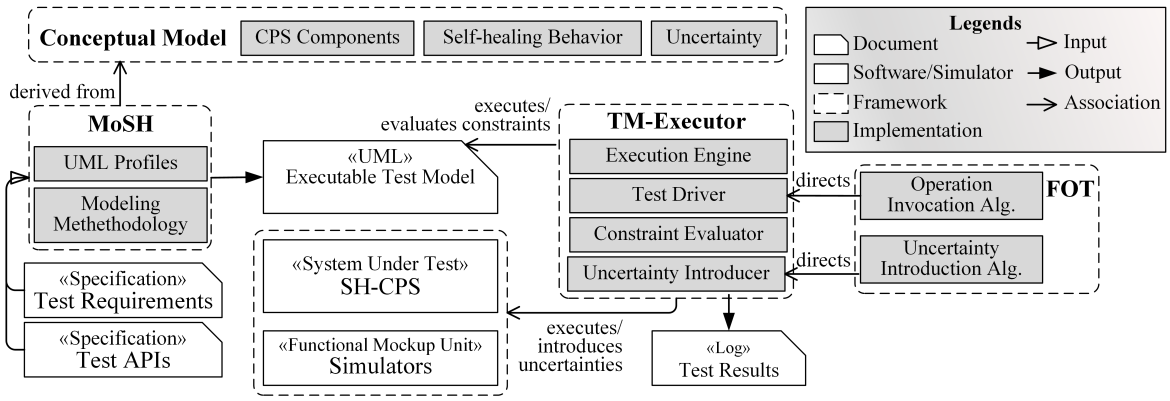


Figure 3 Overview of Executable Model-Based Testing Methodologies

## 4.1 Modeling Foundations

To realize the executable model-based testing, we first need to know how to specify the expected behaviors of an SH-CPS as an executable model. As both SH-CPS and uncertainty are complex concepts, we derived a conceptual model of SH-CPS and uncertainties (Section 4.1.1) from the literature, to understand and capture key components and their relations of SH-CPSs and uncertainties. Based on the conceptual model, a modeling framework — MoSH (Section 4.1.2) was developed. It includes a set of UML stereotypes, datatypes, and a modeling methodology for creating executable test models. Furthermore, a testing framework — TM-Executor (Section 4.1.3) has been implemented to execute the model and perform executable model-based testing.

### 4.1.1 Conceptual Model of SH-CPSs and Uncertainties (CMSU)

The conceptual model is composed of three parts: Components of Cyber-Physical Systems, Self-Healing Behavior, and Uncertainty. Figure 4 presents the key concepts contained in the first two parts. As shown in the figure, a *Cyber Physical System* can be seen as a collection of heterogeneous, distributed *Physical Unit*s. *Controller*s are the cores of *Physical Unit*s. They provide control logic and computation capabilities to *Physical Unit*s. Via *Sensor*s and *Actuator*s, the *Controller*s monitor and control physical processes.

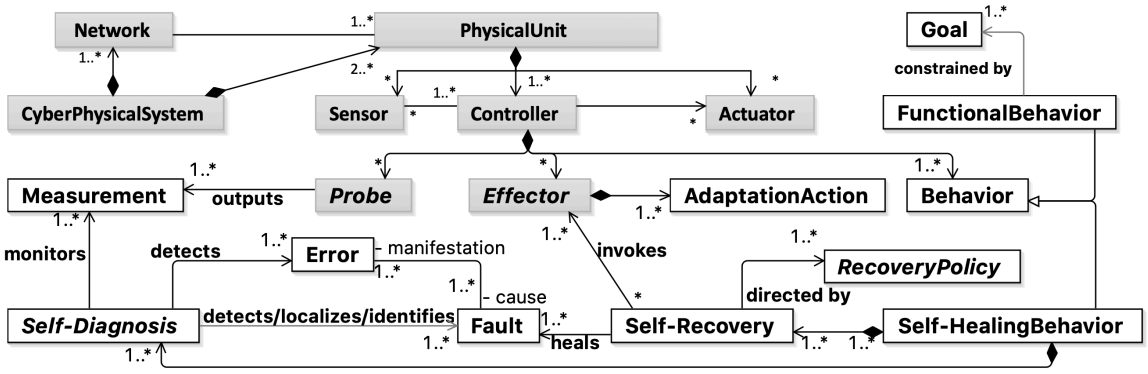


Figure 4 Concepts Relevant to Components of Cyber-Physical Systems and Self-Healing Behavior

For an SH-CPS, a *Controller* has two types of behaviors: 1) *Functional Behavior*s that are designed to fulfill business requirements; and 2) *Self-Healing Behavior*s that use *Probe*s and *Effector*s to help *Functional Behavior*s to achieve their *Goal*s even in the presence of *Fault*s. A *Self-Healing Behavior* consists of *Self-Diagnosis* and *Self-Recovery*. *Self-Diagnosis* is for detecting, localizing, or identifying *Fault*s based on the *Measurement*s from *Probe*s. *Self-Recovery* uses *Effector*s to adapt system behaviors at runtime to recover from detected *Fault*s, directed by *Recovery Policie*s.

Figure 5 presents the key concepts about *Uncertainty*. *Uncertainty* is defined as the lack of knowledge of which value an *Uncertain Feature* will take at a given point in time (*Time Instance*) during execution. *Universe* describes the set of all possible values that an *Uncertain Feature* may take and each possible value is defined as a *Datum*. When we only have qualitative knowledge, the set of possible values (*Datum*s) has to be described qualitatively. A qualitative term is defined as *Category*. A *Membership Function* can be used to specify to what extent a *Datum* belongs to a *Category*. Depending on the available knowledge about the *Uncertainty*, we can use *Probability* or *Possibility Measure* to specify the likelihoods of *Datum*s or *Categori*es.

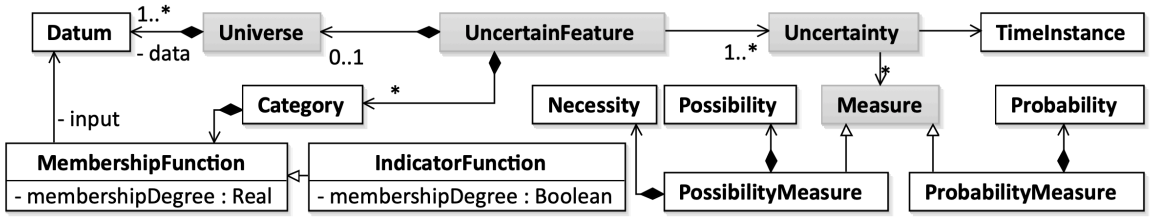


Figure 5 Concepts Relevant to Uncertainty

### 4.1.2 MoSH Modeling Framework

Based on the conceptual model, UML profile of **M**odeling and **A**nalysis of **R**eal-**T**ime **E**mbedded Systems (MARTE) [28], and **U**ML **T**esting **P**rofile (UTP) [29], we implemented a modeling framework — MoSH. It provides a set of stereotypes and datatypes, organized in four UML profiles, to specify the expected behaviors and uncertainties of SH-CPSs, as shown in Figure 6.

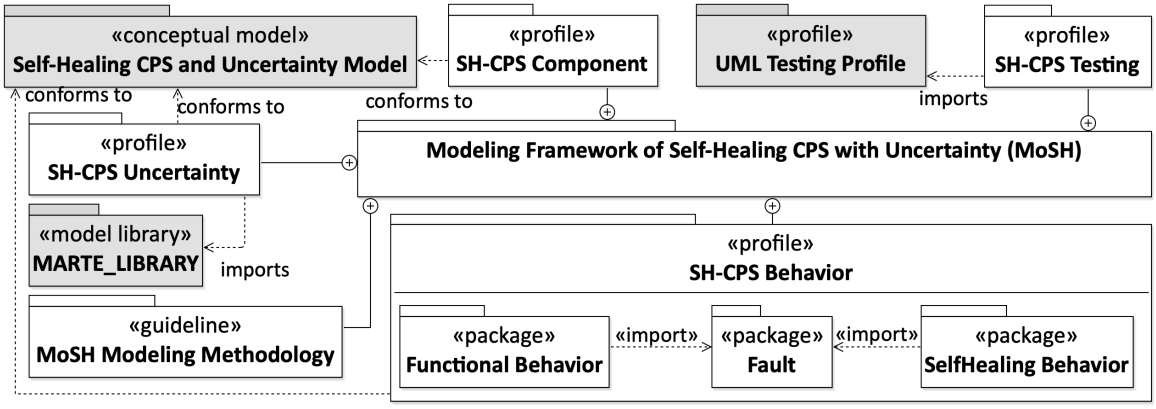


Figure 6 Overview of MoSH

SH-CPS Component profile provides six stereotypes, like «PhysicalUnit» and «Controller», to annotate the role played by each system component. The component is specified as a UML class, with its accessible state variables and testing interfaces captured as class attributes and operations. The expected behaviors of each component are specified as UML state machines. The SH-CPS Behavior profile offers «FunctionalBehavior» and «SelfHealingBehavior» to distinguish the two types of behaviors. For a functional behavior, a *Change Event* stereotyped with «Fault» can be used to specify how a fault is going to affect the behavior, and a *State* stereotyped with «Error» can be used to define the consequent, error state after the fault has occurred. For self-healing behaviors, three stereotypes, «MonitoringState», «FaultIdentifiedState», and «AdaptatingState» were defined to identify the key stages of an self-healing behavior, i.e., self-diagnosis and self-

recovery. For self-diagnosis, the logic of fault detection is specified via the transitions between «MonitoringState»s and «FaultIdentifiedState»s. «MonitoringState» is defined to annotate the states, where the self-healing behavior tries to detect faults based on measurements from probes. «FaultIdentifiedState» denotes that the self-healing behavior has detected, localized, or identified a fault. For self-recovery, the transitions between «FaultIdentifiedState»s and «AdaptatingState»s capture its recovery policy. «AdaptatingState» annotates the states where adaptations are to be performed by the self-healing behavior to recover from the fault. The states defined in state machines should also be precisely defined by state invariants, i.e., OCL constraints on class attributes. During test execution, actual class attributes values can be obtained from the system under test and be used to evaluate the invariants. If an invariant is violated, it means an unexpected behavior is detected.

To precisely define an uncertainty, the SH-CPS Uncertainty profile provides seven datatypes, *U_Boolean*, *U_Integer*, *U_Real*, *U_UnlimitedNatural*, *U_Transition*, *U_String*, and *U_Equation*, to define the set of possible values (i.e., Universe) of an uncertainty. Based on the MATLAB fuzzy library [30], the profile provides six kinds of *Membership Function*s to define a qualitative term (i.e., Category). *Probability Measure* or *Possibility Measure* datatypes are also defined in the profile to specify the measurement of uncertainty.

In the SH-CPS Testing profile, «TestItem» and «TestComponent» are provided to identify the testing target, i.e., controllers of SH-CPSs, and simulated components like sensors and actuators, respectively. «CreateStimulusAction» and «CheckPropertyAction» are defined to capture testing interfaces used for controlling and monitoring systems. «CheckPropertyAction» annotates operations that are used to query the values of state variables, while «CreateStimulusAction» annotates operations that are used to control the behaviors of a system.

With the MoSH modeling framework, we can specify the components, expected behaviors, and uncertainties of an SH-CPS as a set of UML class diagrams and state machines. It forms the test model of the SH-CPS. To make the test model executable, the modeling notations are restricted to the executable subset of UML, defined by fUML [4] and PSSM [5]. We also defined the execution semantics of MoSH stereotypes to enable a stereotyped test model to be executed in a deterministic manner.

### 4.1.3 TM-Executor Framework

Based on existing standards of fUML [4], PSSM [5] and FMI [6], a testing framework — TM-Executor was implemented. It takes charge of executing and testing a system against a test model, simulating the effect of uncertainties, and orchestrating the execution of the test model, the system, and simulators/emulators. Figure 7 shows the key components of TM-Executor.

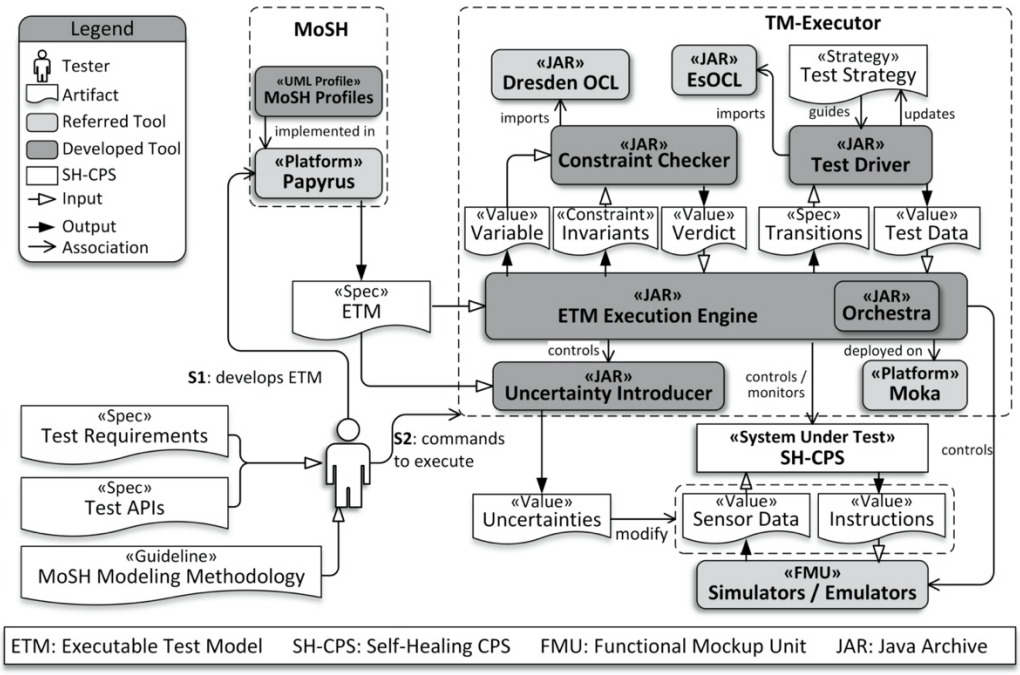


Figure 7 Overview of TM-Executor

The input of TM-Executor is an executable test model (TM), which is created with MoSH modeling framework (Section 4.1.2). In TM-Executor, an Execution Engine executes the TM, the SH-CPS under test and simulators of sensors and actuators together. During execution, the Execution Engine periodically obtains the values of state variables from the system under test, and a Constraint Checker uses the values to evaluate the invariants defined in the model. If any invariant is evaluated to be false, it means the system fails to behave consistently with the test model and a fault is detected.

To drive the execution, a Test Driver was implemented to generate operation invocations, based on the triggers of outgoing transitions of the current active state. The invocations are performed on both the system and the test model to trigger them switching their states. Meanwhile, an Uncertainty Introducer generates a value for each specified uncertainty and

uses the value to modify sensor data measured by sensors or instructions sent to actuators to mimic the effect of uncertainty.

The test model, SH-CPS, and simulators/emulators have to be executed coordinately to fulfill the executable model-based testing. These three kinds of executable objects are typically implemented with different modeling/programming paradigms. To orchestrate the executions of the hybrid objects, an Orchestra was implemented based on the FMI standard [6]. It takes charge of propagating values through the executable objects and synchronizing their executions.

## 4.2 Fragility-Oriented Testing

With EM-Executor, we can dynamically test an SH-CPS against a test model by executing them together, while a testing algorithm is needed to decide how to choose the operation invocations and uncertainty values to drive the execution and detect faults effectively. Particularly, two tasks have to be addressed to solve the testing problem. The first task is to find the optimal sequence of operation invocations for a given test model to maximize the chance of detecting faults. The second task is to find the optimal values of uncertainties that can make the SH-CPS under test violate an invariant defined in the test model, when the system is handling the selected optimal sequence of operation invocations.

To resolve the two tasks, a fragility-oriented testing approach was proposed. Here, the fragility is defined as a distance indicating how likely an invariant is to be violated. By taking the fragility as a reward, we devised two reinforcement learning based algorithms to find the operation invocations and uncertainty values that can work together to reveal a fault.

To resolve the first task, the first algorithm aims to find the optimal sequence of transitions that can lead to the highest fragility, for a given set of state machines defined in a test model. To achieve this objective, the algorithm applies Q-learning [8], a reinforcement learning algorithm, to estimate the highest fragility that can be reached after a given transition is triggered. By focusing on choosing the transitions with high estimated fragility, the algorithm can gradually find the optimal sequence of transitions and generate a sequence of operation invocations that can trigger the transitions and reach the highest fragility.

To resolve the second task, the second algorithm uses an artificial neural network as the policy of selecting uncertainty values. The neural network takes the state of the system under test as input, and outputs a value for each uncertainty. The value is used as the mean value of a truncated normal distribution, with its variance fixed at a constant positive value. That is, the outputs of the neural network determine a probability distribution for each uncertainty. By sampling from the distribution, uncertainty values can be selected to test an SH-CPS. To effectively find faults, we need to optimize the policy so that the uncertainty values generated from the policy can increase the fragility of the system under test and make the system likely to fail. To do so, the algorithm keeps on selecting uncertainty values following the policy. When it observes a sequence of uncertainty values reaches a higher fragility, it takes the gradient descent algorithm [31] to update the neural network, so as to increase the selection probability of this sequence of uncertainty values. The algorithm keeps on this search process until reaching the maximum iterations.

## 4.3 Empirical Study

Several existing reinforcement learning algorithms [32] can be used in the fragility-oriented testing approach. However, there is no sufficient evidence showing which reinforcement learning algorithms are the best to be used for testing SH-CPS under uncertainty. To this end, we conducted an empirical study to evaluate the performance of 14 combinations of reinforcement learning algorithms for testing six SH-CPSs under dozens of identified uncertainties. As the task of selecting operation invocations is different from the task of selecting uncertainty values, we selected two sets of algorithms to perform them. Specifically, we applied two value function learning based algorithms, Action-Reward-State-Action (SARSA) [8] and Q-learning [8] for selecting operation invocations, and seven policy optimization based algorithms, Asynchronous Advantage Actor-Critic (A3C) [33], Actor-Critic method with Experience Replay (ACER) [34], Proximal Policy Optimization (PPO) [35], Trust Region Policy Optimization (TRPO) [36], Actor-Critic method using Kronecker-factored Trust Region (ACKTR) [37], Deep Deterministic Policy Gradient (DDPG) [38], and Uncertainty Policy Optimization (UPO) [39], for learning the policy of selecting values for uncertainties.

In this empirical study, we first tuned the hyperparameters of these algorithms by applying them to test three of the selected SH-CPSs with diverse complexities. Afterward,

we applied the 14 combinations of algorithms with the tuned hyperparameters to test all six SH-CPSs. Based on the testing results, we evaluated the effectiveness, efficiency and scalability of each combination, by testing coverage, number of detected faults, average time spent to detect a fault, time and space cost used to perform the test.

# 5 Summary of Results

This section summarizes the key evaluation results of the modeling foundations, fragility-oriented testing, and empirical study, corresponding to three papers submitted as a part of this thesis.

## 5.1 Modeling Foundations (Paper A)

*"Modeling Foundations for Executable Model-Based Testing of Self-Healing Cyber-Physical Systems" T. Ma, S. Ali, and T. Yue. Journal of Software & Systems Modeling (SOSYM). DOI: 10.1007/s10270-018-00703-y*

In this paper, a modeling framework of SH-CPS (MoSH) was created based on a conceptual model of SH-CPS and uncertainty (CMSU). Besides, it also provides a testing framework (TM-Executor) to test an SH-CPS against a test model, specified with MoSH, by co-executing them together.

First, the conceptual model of SH-CPS and uncertainty was evaluated to check if the model can correctly cover the concepts and their relations identified from nine SH-CPSs, obtained from the literature. Based on the specification of the nine systems, we first identified their main components. For each component, we captured its behaviors and the environmental uncertainties that may affect these behaviors. For self-healing behaviors, we further identified their strategies to detect and recover from faults. The identified components, behaviors and uncertainties were manually mapped to the concepts and relationships in the conceptual model. Figure 8 presents the process, in which we verified and improved the conceptual model by this mapping. Initially, we derived the conceptual model (*CMSU V.1*) from the literature (Activity *A1* in Figure 22). To evaluate its quality, we identified SH-CPS related concepts as well as their relationships (*Cons. & Rels. from CSs. V.1*), from the nine SH-CPSs' specifications (Activity *A2.1*). *Cons. & Rels. from CSs. V.1* contains necessary entities required to specify self-healing behaviors and uncertainties of

an SH-CPS. For each identified concept or relationship, we tried to manually find a counterpart in *CMSU V.1* (Activity *A2.2*). If the counterpart is missing, we further investigated if the extracted concept or relationship is correctly identified. In case that it was correct, *CMSU V.1* was revised to cover the missing concept. Otherwise, the incorrectly identified concept or relationship was fixed. After *A2.2*, we created a new version of the extracted concepts and relationships, i.e., *Cons. & Rels. from CSs. V.2*. At last, the refined conceptual model (*CMSU V.2*) was further refined by *A3* via a mapping from *Cons. & Rels. from CSs. V.2* to *CMSU V.2*. The final obtained *CMSU V.3* managed to correctly cover all identified concepts and relations.

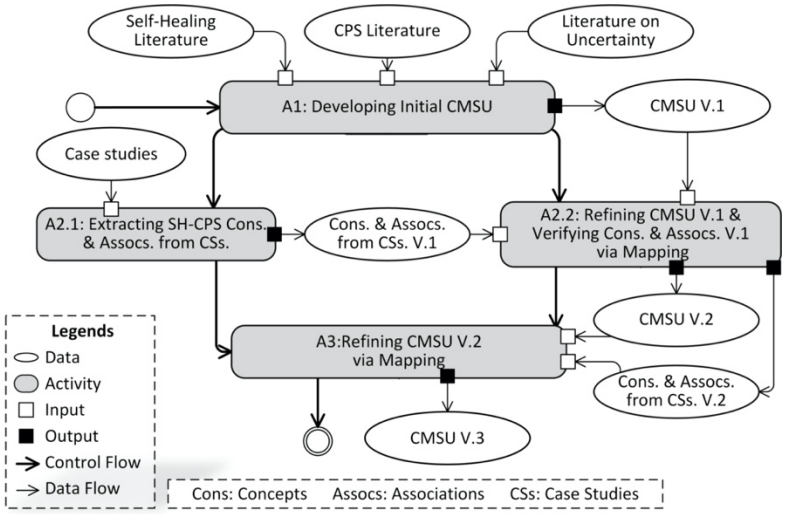


Figure 8 Process to Develop CMSU

Afterward, we assessed if MoSH provides a cost-effective way of creating executable test models, by applying MoSH to create test models for three SH-CPSs. To measure the extra modeling effort required for applying MoSH, we calculated the total number of model elements and the percentage of stereotyped model elements. For the applicability, we checked the numbers of functional behaviors, self-healing behaviors, self-diagnosis behaviors, self-recovery behaviors, and uncertainties that can be specified with MoSH. On average, 206 model elements were used to build an executable test model for an SH-CPS. 16 percent of the model elements were stereotyped with MoSH. This means that it needed an additional 16% modeling effort to apply MoSH to create the executable test models, as compared with applying standard UML notations, for the selected SH-CPSs. In total, 20 functional behaviors, 11 self-healing behaviors, 11 self-diagnosis behaviors, 17 self-recovery behaviors and 17 uncertainties were specified for the three systems. This

demonstrates the applicability of MoSH to specify executable test models for three diverse SH-CPSs. This effort gives us evidence that MoSH is capable of modeling different SH-CPSs to support uncertainty-aware executable model-based testing.

Furthermore, the performance of TM-Executor was evaluated to determine if it is feasible to apply the framework to test a complex SH-CPS. For this evaluation, we applied TM-Executor to test an SH-CPS against its test model, with operation invocations and uncertainty values randomly selected. We assessed how much time is required by TM-Executor to execute a test model, generate test data, evaluate constraints, and introduce uncertainties. On average, it took 5.6 seconds for traversing a transition, 39 milliseconds for test data generation, and less than one millisecond for exiting or entering a state, executing an operation, evaluating a constraint or generating an uncertainty value. The result indicates the time taken by TM-Executor to perform testing activities was relatively small, and thus it is practicable to apply TM-Executor to perform the executable model-based testing.

## 5.2 Fragility-Oriented Testing (Paper B)

*"Testing Self-Healing Cyber-Physical Systems under Uncertainty: A Fragility-Oriented Approach" T. Ma, S. Ali, T. Yue, and M. Elaasar. Software Quality Journal (SQJ). DOI: 10.1007/s11219-018-9437-3*

In this paper, a fragility-oriented testing approach was devised to effectively detect faults in SH-CPSs under uncertainty. The approach is comprised of two reinforcement learning based algorithms: Fragility-Oriented Operation Invocation (FOOI)[2] for invoking operations and Uncertainty Policy Optimization (UPO) for introducing uncertainties. The two algorithms utilize the fragility, obtained from test executions, to learn the optimal policies of invoking operations and introducing uncertainties, so as to detect faults in the most effective manner.

To evaluate the effectiveness of the two proposed algorithms, we chose a coverage-oriented testing algorithm (COT) as the benchmark for operation invocations, and a

[2] The algorithm is named as Fragility-Oriented Testing in Paper B. To distinguish the overall fragility-oriented testing approach and the algorithm for selecting operation invocations, the algorithm is named as Fragility-Oriented Operation Invocation (FOOI) in this section.

random approach (R) for introducing uncertainties. In total, we obtained four approaches: FOOI+UPO, FOOI+R, COT+UPO, COT+R. They were applied to test three SH-CPSs to check which one detects more faults. The number of detected faults is not only determined by the fault detection ability of the algorithms, but also affected by the amount of testing time and the variation range of each uncertainty. To reduce the effect of testing time and scale of uncertainty variation, we chose three settings for each of them. The three testing times are 72, 144, and 216 hours. This allows each testing approach to approximately execute an SH-CPS with its test model 500, 1000, and 1500 times to find faults. Meanwhile, we chose three scales of uncertainty variation: 80%, 100%, and 120%. 100% represents the standard variation ranges, obtained from the production specification of sensors and actuators. 80% (120%) means reducing (increasing) the ranges by 20 percent. For each case study and each test setting, we collected the number of faults detected by each testing approach. It turns out FOOI+UPO detected more faults than the other three approaches for all cases. We further conducted Mann-Whitney U test with a significant level of 0.05 to determine the significance of the differences. Results of the statistic tests show that FOOI+UPO significantly outperformed the others in most cases. Only when the testing time is 72 hours and the uncertainty scale is 80%, there is no significant difference among the four testing approaches. As the scale of uncertainty is low, the four testing approaches only detected few faults within 72 hours.

## 5.3 Empirical Study (Paper C)

*"Testing Self-Healing Cyber-Physical Systems under Uncertainty with Reinforcement Learning: An Empirical Study" T. Ma, S. Ali, and T. Yue. Journal of Empirical Software Engineering (EMSE). DOI: 10.1007/s10664-021-09941-z.*

This paper presents an empirical study, in which the effectiveness, efficiency, and scalability of 14 combinations of reinforcement learning algorithms were evaluated for testing SH-CPSs under uncertainty. The 14 combinations are composed of two value function learning based methods (Action-Reward-State-Action (SARSA) [8] and Q-learning [8]) for operation invocations and seven policy optimization based algorithms (Asynchronous Advantage Actor-Critic (A3C) [33], Actor-Critic method with Experience Replay (ACER) [34], Proximal Policy Optimization (PPO) [35], Trust Region Policy

Optimization (TRPO) [36], Actor-Critic method using Kronecker-factored Trust Region (ACKTR) [37], Deep Deterministic Policy Gradient (DDPG) [38], and Uncertainty Policy Optimization (UPO) [39]) for introducing uncertainties. These algorithms were applied to test six SH-CPSs. Three of them are real-world case studies, and the others are from the literature.

For effectiveness, all 14 combinations of algorithms managed to cover all states and transitions of most case studies, except the most complex one. The p-values of the Kruskal-Wallis test in terms of the state and transition coverages for all the testing tasks are greater than 0.1, therefore, indicating no significant difference among the 14 approaches regarding the coverages. In total, 41 faults were detected in the six SH-CPSs. These 41 faults correspond to 41 states in which the invariants of the states were violated when the six systems were being tested with simulated sensors and actuators. Q-learning + UPO managed to detect the most faults. On average, it detected 3.4 faults for each case study. SARSA + UPO performed slightly worse, with 3.3 faults detected averagely. In contrast, the other 12 approaches only detected 1.7 faults, on average. In over 239 (out of 300) testing jobs, Q-learning + UPO significantly outperformed the other 12 testing approaches, except SARSA + UPO. Q-learning + UPO and SARSA + UPO performed equally in 279 jobs; SARSA + UPO beat Q-learning + UPO in 2 jobs and Q-learning + UPO was superior in the other 19 jobs.

Regarding efficiency, Q-learning + UPO took the least amount of time to detect a fault, since all testing approaches took a similar amount of time and Q-learning + UPO detected the most faults. Averagely, Q-learning + UPO took 64.5 hours to detect a fault, which is less than half of the average time taken by Q-learning + A3C (the least efficient approach) for fault detection. On average, Q-learning + UPO took 52% less time than the other approaches to detect a fault.

Concerning scalability, we assessed the tendencies of time and space costs of the 14 testing approaches, as the number of states and transitions of the system under test increases. It turns the more states and transitions an SH-CPS has, the more time and space a testing approach took to learn the optimal policy of invoking operations. In contrast, the time and space costs of learning the policy of selecting uncertainty values remain in the same order of magnitude for testing SH-CPSs with diverse complexities. This is due to the difference of the two types of reinforcement learning algorithms used to learn these two

policies. Value function learning based algorithms (SARSA and Q-learning) were used for operation invocations and policy optimization based algorithms were used for introducing uncertainties. For value function learning based algorithms, they have to save the highest fragility that can be reached after triggering each transition. As the number of transitions increases, such methods will take more space and time to store and process the fragility values. Alternatively, policy optimization based algorithms use artificial neural networks to approximate their policies and value functions, the computational costs of these algorithms are determined by the architecture of the neural networks and the optimizer to improve the neural networks.

# 6 Future Directions

This thesis provides a complete executable model-based testing solution for testing SH-CPS under uncertainty, including a modeling framework (MoSH) to create executable test models, a testing framework (TM-Executor) to test an SH-CPS against a test model, and a fragility-oriented testing approach to effectively detect faults. This section discusses possible future directions of these works to further refine the executable model-based testing approach.

For the modeling framework, it has been designed to create executable test models for a specific type of system, i.e., SH-CPS. A more general modeling framework could be derived from MoSH to support modeling a broader range of systems. In addition, more case studies and a larger scale of applications are needed to further validate and improve the applicability of MoSH.

Regarding the testing framework, it has been implemented based on the standard of FMI [6] to co-execute hybrid executable objects, like the test model, software of SH-CPSs, and simulators of sensors and actuators. As SH-CPSs are complex, it is computationally expensive and time-consuming to execute them. A distributed testing framework could be built in the future to allocate more computation resources to perform the co-execution and reduce the time spent on execution.

The proposed fragility-oriented testing approach makes it possible to apply various reinforcement learning algorithms to perform testing. As reinforcement learning is still an actively evolving field and more advanced reinforcement learning algorithms are emerging, the newly devised algorithms can be tested in this approach to possibly further enhance the fault detection ability.

# 7 Conclusion

Self-healing is becoming a critical feature of Cyber-Physical Systems (CPSs). It enables CPSs to detect faults and apply proper adaptations to recover from the faults at runtime. We refer CPSs with the self-healing features as Self-healing CPSs (SH-CPSs). Besides recovery, SH-CPSs have to deal with various uncertainties, like measurement errors from sensors and actuation deviations from actuators. To assess the dependability of SH-CPSs, it is necessary to test if an SH-CPS can still behave as expected under these uncertainties.

This thesis provides an executable model-based testing approach to solve this testing problem. The whole approach is composed of (1) CMSU, a conceptual model of SH-CPS and uncertainty, (2) MoSH, a modeling framework of SH-CPS to create executable test models, (3) TM-Executor, which enables co-execution of the test model and SH-CPS, (4) Fragility-Oriented Testing, which takes fragility as the rewards of reinforcement learning algorithms to effectively detect faults. At last, an empirical study was conducted to find the optimal one among 14 combinations of reinforcement learning algorithms for testing SH-CPSs under uncertainty.

By evaluating these works with diverse SH-CPSs, we demonstrated that (1) MoSH provides sufficient modeling notations and methodology to specify executable test models for SH-CPSs; (2) the overhead of applying TM-Executor to perform the executable model-based testing is small, and it is practical to apply TM-Executor to test SH-CPSs; (3) the fault detection ability of the fragility-oriented testing approach is significantly higher than random testing and coverage-oriented testing. Reinforcement learning algorithms have shown competence in detecting faults in SH-CPSs under uncertainty. Based on the results of the empirical study, we found that the combination of Q-learning and Uncertainty Policy Optimization algorithms managed to detect the most faults in six SH-CPSs. On average, they managed to discover two times more faults than the other 13 combinations of reinforcement learning algorithms.

# 8 References for Summary


[1] E. A. Lee, "Cyber physical systems: Design challenges," in *11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008 2008: IEEE, pp. 363-369.

[2] T. Bures *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," *ACM SIGSOFT Software Engineering Notes,* vol. 40, no. 6, pp. 28-32, 2015.

[3] T. Ma, S. Ali, and T. Yue, "Modeling foundations for executable model-based testing of self-healing cyber-physical systems," *Software and Systems Modeling,* 2018. [Online]. Available: DOI: 10.1007/s10270-018-00703-y.

[4] *Semantics Of A Foundational Subset For Executable UML Models V1.2.1*, OMG, 2016.

[5] *Precise Semantics Of UML State Machines (PSSM)*, OMG, 2017.

[6] T. Blochwitz *et al*., "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proceedings of the 9th International MODELICA Conference*, 2012, no. 076, pp. 173-184.

[7] T. Ma, S. Ali, T. Yue, and M. Elaasar, "Testing Self-Healing Cyber-Physical Systems under Uncertainty: A Fragility-Oriented Approach," *Software Quality Journal,* 2018.

[8] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction* (no. 1). MIT press Cambridge, 1998.

[9] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* vol. 42, no. 1, pp. 101-111, 2012.

[10] X. Zhou, X. Gou, T. Huang, and S. Yang, "Review on Testing of Cyber Physical Systems: Methods and Testbeds," *IEEE Access,* vol. 6, pp. 52179-52194, 2018.

[11] X. Zheng and C. Julien, "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*, 2015: IEEE Press, 2015, pp. 15-18.

[12] S. Ali, H. Lu, S. Wang, T. Yue, and M. Zhang, "Uncertainty-Wise Testing of Cyber-Physical Systems," in *Advances in Computers*, vol. 107: Elsevier, 2017, pp. 23-94.

[13] R. De Lemos *et al*., "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*: Springer, 2013, pp. 1-32.

[14] A. Computing, "An architectural blueprint for autonomic computing," *IBM Publication,* 2003.

[15] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model," in *Modelling Foundations and Applications: 12th European Conference, ECMFA* 2015: Springer 2015.

[16] A. J. Ramirez, A. C. Jensen, and B. H. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* 2012: IEEE, 2012, pp. 99-108.

[17] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
[18] K. Gama and D. Donsez, "Deployment and activation of faulty components at runtime for testing self-recovery mechanisms," *ACM SIGAPP Applied Computing Review,* vol. 14, no. 3, pp. 44-54, 2014.
[19] J. Park, S. Lee, T. Yoon, and J. M. Kim, "An autonomic control system for high-reliable CPS," *Cluster Computing,* vol. 18, no. 2, pp. 587-598, 2015.
[20] D. Drones, "ArduCopter," in *http://ardupilot.org/copter/*, ed, 2010.
[21] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software testing, verification and reliability,* vol. 13, no. 1, pp. 25-53, 2003.
[22] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner, and B. Meyer, "On the number and nature of faults found by random testing," *Software Testing, Verification and Reliability,* vol. 21, no. 1, pp. 3-28, 2011.
[23] Q. Xie, "Developing cost-effective model-based techniques for GUI testing," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 997-1000.
[24] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association,* vol. 47, no. 260, pp. 583-621, 1952.
[25] O. J. Dunn, "Multiple comparisons using rank sums," *Technometrics,* vol. 6, no. 3, pp. 241-252, 1964.
[26] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the royal statistical society. Series B (Methodological),* pp. 289-300, 1995.
[27] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132, 2000.
[28] *Profile for modeling and analysis of real-time and embedded systems (MARTE)*, OMG, 2011.
[29] *UML Testing Profile*, OMG, 2017. [Online]. Available: https://www.omg.org/spec/UTP/About-UTP/
[30] S. Sivanandam, S. Sumathi, and S. Deepa, *Introduction to fuzzy logic using MATLAB*. Springer, 2007.
[31] R. Pascanu and Y. Bengio, "Revisiting natural gradient for deep networks," *arXiv preprint arXiv:1301.3584,* 2013.
[32] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866,* 2017.
[33] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, 2016, pp. 1928-1937.
[34] Z. Wang *et al.*, "Sample efficient actor-critic with experience replay," *arXiv preprint arXiv:1611.01224,* 2016.
[35] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347,* 2017.
[36] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015: PMLR, 2015, pp. 1889-1897.

[37] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Advances in neural information processing systems*, 2017, 2017, pp. 5279-5288.

[38] T. P. Lillicrap *et al*., "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971,* 2015.

[39] T. Ma, S. Ali, T. Yue, and M. Elaasar, "Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach," *Software Quality Journal,* vol. 27, no. 2, pp. 615-649, 2019.

# Part II

Papers

# Paper A

# Modeling Foundations for Executable Model-Based Testing of Self-Healing Cyber-Physical Systems

Tao Ma, Shaukat Ali, Tao Yue

Journal of Software & Systems Modeling (SOSYM). DOI: 10.1007/s10270-018-00703-y

# Abstract

Self-healing Cyber-Physical Systems (SH-CPSs) detect and recover from faults by themselves at runtime. Testing such systems is challenging due to the complex implementation of self-healing behaviors and their interaction with the physical environment, both of which are uncertain. To this end, we propose an executable model-based approach to test self-healing behaviors under environmental uncertainties. The approach consists of a **M**odeling Framework **o**f **SH**-CPSs (MoSH) and an accompanying **T**est **M**odel **Executor** (TM-Executor). MoSH provides a set of modeling constructs and a methodology to specify executable test models, which capture expected system behaviors and environmental uncertainties. TM-Executor executes the test models together with the systems under test, to dynamically test their self-healing behaviors under uncertainties. We demonstrated the successful application of MoSH to specify 11 self-healing behaviors and 17 uncertainties for three SH-CPSs. The time spent by TM-Executor to perform testing activities was in the order of milliseconds, though the time spent was strongly correlated with the complexity of test models.

**Keywords**: Cyber-Physical Systems, Self-healing, Uncertainty, Model Execution, Model-Based Testing

# 1 Introduction

Self-healing[3] is becoming an important feature of **C**yber-**P**hysical **S**ystems (CPSs) [1]. A **S**elf-**H**ealing **C**yber-**P**hysical **S**ystem (SH-CPS) can perceive that it is not operating correctly and, without human intervention, makes necessary changes to its architecture or behaviors to restore itself to a normal state [2]. Despite the benefits offered by self-healing, there is still a need for novel testing methods to gain confidence in decisions made by such behaviors [3]. Since such systems often operate in an unpredictable environment [4], they need to be tested under environmental uncertainties. Examples of such uncertainties

[3] The term of self-healing originates from the IBM's vision of autonomic computing [3] and self-healing is one of the so-called self-* properties. Self-healing is tightly related to "fault-tolerant", but not all fault-tolerant mechanisms can be seen as self-healing behaviors.

include errors in data measured by sensors, actuation deviations from actuators, and latency in networks.

To effectively detect faults in an SH-CPS under uncertainties, we propose to test the system against an executable test model directly. By executing the system and the test model together, testing stimuli are dynamically selected from the model at runtime, directed by a test strategy and followed by executing the stimuli on the system and the model. Meanwhile, uncertainties specified in the test model are introduced via simulators/emulators of the environment. As the result of executing the stimuli, the test model and the system switch to their subsequent states. These states are compared to determine if the system behaves as expected. Such comparison also reveals how likely the system's behaviors are going to deviate from the modeled behaviors. The likelihood can be used as a heuristic to find the optimal sequence of stimuli that can help to detect faults efficiently.

Realizing such an executable model-based testing approach requires three enablers. First, a modeling framework is required to facilitate the specification of executable test models. Though there exist modeling solutions [5-9], it still lacks modeling notations and methodologies for specifying self-healing behaviors and uncertainties. Second, a test execution environment is needed to execute the test models, generate testing stimuli, introduce uncertainties, and test systems against the models. Third, it requires a testing strategy that can take advantage of the runtime information obtained from the execution to find the optimal sequence of stimuli and effectively detect faults.

As the first step to realize the executable model-based approach, we propose a modeling framework and an accompanying execution environment in this paper. We first developed a conceptual model to capture necessary SH-CPS and uncertainty concepts, based on which, we developed an uncertainty-aware **M**odeling framework **o**f **SH**-CPSs (MoSH) to assist the specification of executable test models. The modeling notations of MoSH are restricted to the executable subset of UML, i.e., the **F**oundational **UML** Subset (fUML) standard [10]. The fUML and **P**recise **S**emantics Of UML **S**tate **M**achines (PSSM) standards [11] define the execution semantics for the subset of UML and thus enable the test model to be directly executed without model transformation to executable test cases. Based on the two standards, we developed a **T**est **M**odel **Executor** (TM-Executor) as the

execution environment. Testing strategies are not covered in this paper, whereas they can be found in our recently published paper [12].

We evaluated MoSH and TM-Executor from three aspects: 1) quality of the conceptual model, based on which MoSH was implemented, 2) applicability of MoSH to build executable test models, and 3) performance of TM-Executor regarding the time required to perform test execution. The evaluation demonstrated that the conceptual model managed to cover 832 instances of concepts identified in the nine SH-CPSs. We successfully applied MoSH to specify 11 self-healing behaviors and 17 uncertainties, identified in three out of the nine systems. On average, 16% of model elements were stereotyped with the MoSH profiles. Regarding performance, the time spent by TM-Executor to perform testing activities was in the order of milliseconds. The complexity of guards significantly affected the time cost of TM-Executor for generating test data. For the most complex guard, TM-Executor took a maximum of 0.27 seconds to generate test data. However, the time spent by TM-Executor to evaluate an invariant or generate an uncertainty value was not significantly affected by the complexity of invariants and different ways of generating uncertainty values.

The rest of this paper is organized as follows: Section 2.2 introduces the background, followed by a running example given in Section 3. Section 4 presents the conceptual model of SH-CPSs and uncertainties. MoSH and its associated modeling methodology are explained in Section 4.1.2. Section 0 presents the main components of TM-Executor. Section 7 presents the evaluation. Related work is discussed in Section 8, and we summarize the paper and future work in Section 9.

# 2 Background

As the proposed executable model-based testing is an extension of model-based testing, we compare the two approaches in Section 2.1. In our work, we applied probability, possibility, and fuzzy set theories to measure uncertainties. As the fuzzy set theory is not commonly known, we introduce it in Section 2.2.

## 2.1 Model-Based Testing versus Executable Model-Based Testing

Figure 9 highlights the differences between the two testing approaches. In model-based testing, a system is tested for checking its conformance to a test model capturing the

system's expected behaviors. Guided by a testing strategy, test cases are generated to cover a finite number of paths that traverse through the model. Afterward, the test cases are executed on the system for detecting faults. In contrast, executable model-based testing directly tests a system against an *executable* test model, by executing the model and the system together, sending them the same stimuli and comparing their consequent states. Based on the executable test model, a testing strategy is used to select the stimuli at runtime to drive the execution. Information about the system's actual behaviors can be obtained from the execution and aids the testing strategy to find the optimal sequence of stimuli for fault detection.

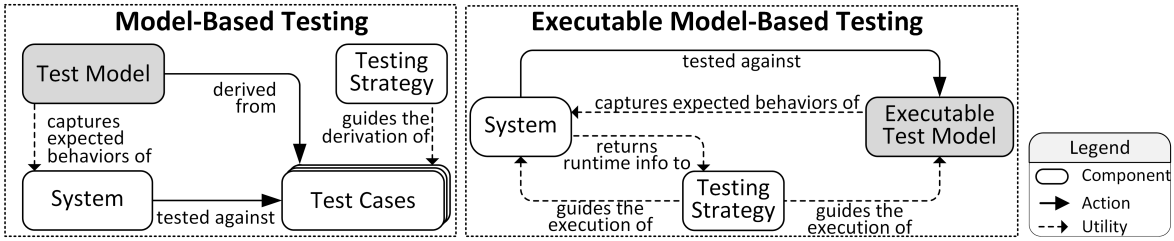


Figure 9 Model-Based Testing versus Executable Model-Based Testing

## 2.2 Fuzzy Set Theory

Uncertainty may originate from various sources in SH-CPSs, and not all sources of uncertainty have the same characteristics. From the perspective of knowledge and the range of uncertainty, uncertainty can be classified into reducible and irreducible uncertainty. From another perspective, uncertainty can be classified as aleatory or epistemic [4]. Aleatory uncertainty is caused by randomness and is usually measured by the probability theory. However, epistemic uncertainty is due to a lack of knowledge and fuzzy sets can be applied to quantify incomplete or imprecise knowledge. The members of the fuzzy set are defined by a membership function, which assigns each possible member a value between 0 and 1, representing the membership degree of the member to the fuzzy set. For instance, the noise value of a GPS sensor can be expressed as *Low*, *Medium*, and *High*. As the boundaries of the three types are not precisely defined, a noise value may partially belong to multiple types. With membership functions, one can mathematically specify the partial belongings. Thus, the fuzzy set theory provides a way to quantify uncertainty, in case testers' knowledge about the uncertainty is unclear.

# 3 Running Example

The running example is an unmanned aerial vehicle control system—**R**emotely operated **A**erial **M**odel **A**utopilot (RAMA)[4] [13]. RAMA consists of two main components (Figure 10): a ***G**round **C**ontrol **S**tation* (*GCS*) and a *Drone*. A human pilot uses the *GCS* to maneuver the *Drone*. Based on the position and terrain information provided by a *PositionLocationUnit* and a *TerrainDatabase*, a *NavigationUnit* calculates a desired flight orientation and controls *Servo*s accordingly to perform a movement.

RAMA aims to prevent the *Drone* from crashing even if one or more components fail to work. To achieve this objective, Holub et al. [13] realized a set of self-healing behaviors to handle faults during the flight. For instance, if the connection between the *Drone* and *GCS* is broken, the *NavigationUnit* detects this fault occurrence via the absence of heartbeats and automatically directs the *Drone* to fly back to its launch position. Another example is the self-healing behavior for a *Servo* fault; when one of the four *Servo*s stops working, the fault is identified by comparing expected and actual orientations of the *Drone* and then the *NavigationUnit* switches to control three dimensions (pitch, roll, and throttle) with the fourth dimension (yaw) uncontrolled, to maintain the flight.

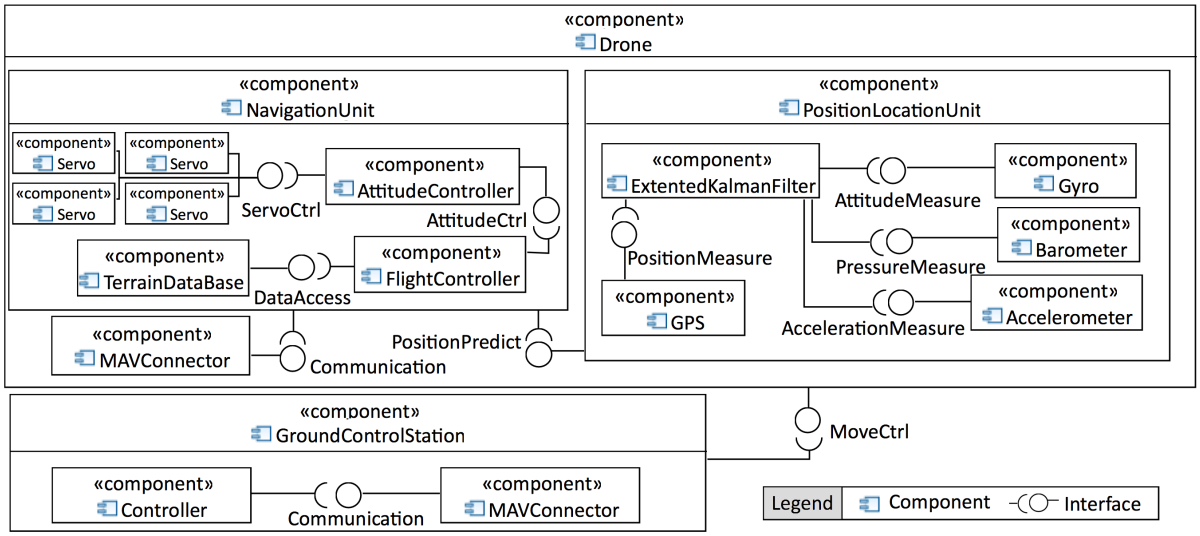


Figure 10 UML Component Diagram of RAMA (Partial)

Besides component faults, environmental uncertainties are another factor that impacts the operation of RAMA, such as measurement uncertainties from sensors and uncertain

[4] The reason to select RAMA as the running example is that RAMA is an example of SH-CPS and contains a set of self-healing behaviors that are affected by uncertainties.

actuation deviations from actuators. To keep the flight stable under such uncertainties, the *NavigationUnit* uses an adaptive control strategy to constantly adjusts control signals for the *Servo*s based on the *Drone*'s position and orientation estimated by the *PositionLocationUnit*.

# 4 Conceptual Model of SH-CPSs and Uncertainties (CMSU)

To realize executable model-based testing, we need a modeling solution to create executable test models. Since CPS, self-healing, and uncertainty are complex concepts, it is important to identify and understand their main components to systematically derive the modeling solution [14-17]. To fulfill this task, we derived a **C**onceptual **M**odel of **S**H-CPSs and **U**ncertainties (CMSU) from the literature. This section presents the three parts of CMSU: Components of Cyber-Physical Systems (Section 4.1), Self-Healing Behavior (Section 4.2), and Uncertainty (Section 4.3). Definition of each concept is provided in a corresponding technical report [18].

## 4.1 Components of Cyber-Physical Systems

A *CyberPhysicalSystem* can be seen as a collection of heterogeneous, distributed *PhysicalUnits* that work together to control or monitor *PhysicalProcesses* (Figure 11). For example, the *GCS* cooperates with the *Drone* to control the flight process (Section 3). *Controller*s are the cores of *PhysicalUnit*s (e.g., the control units in the running example). They provide control logic and computation capabilities to *PhysicalUnit*s and communicate with each other via *Network*s. Relying on *Sensor*s and *Actuator*s, they monitor and control the *State*s of *PhysicalProcess*es.

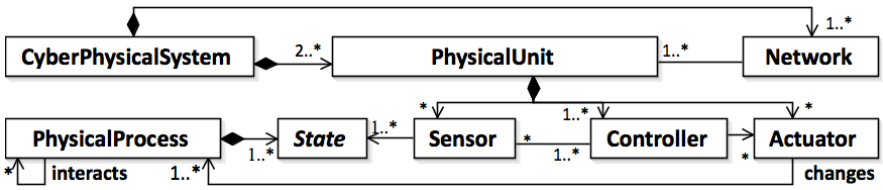

Figure 11 Concepts Relevant to Cyber-Physical System

## 4.2 Self-Healing Behavior

For an SH-CPS, a *Controller* has two types of behaviors: 1) *FunctionalBehavior*s that are designed to fulfill business requirements (specified as *Goal*s of the behaviors); and 2) *Self-*

*HealingBehavior*s that use *Probe*s and *Effector*s to help *FunctionalBehavior*s to achieve their *Goal*s even in the presence of *Fault*s (Figure 12). *Probe*s and *Effector*s are two types of interfaces that are used to inquire a controller's *State*s and adjust its *Behavior*s respectively. In the running example, RAMA is equipped with probes for checking variations of speed and orientation, and effectors that are used to switch control modes. A *Self-HealingBehavior* consists of *Self-Diagnosis* and *Self-Recovery*. *Self-Diagnosis* is for detecting, localizing, or identifying *Fault*s based on the *Measurement*s from *Probe*s. *Self-Recovery* is for recovering from *Fault*s, directed by *RecoveryPolicies*. The subsections below elaborate the two phases in details.

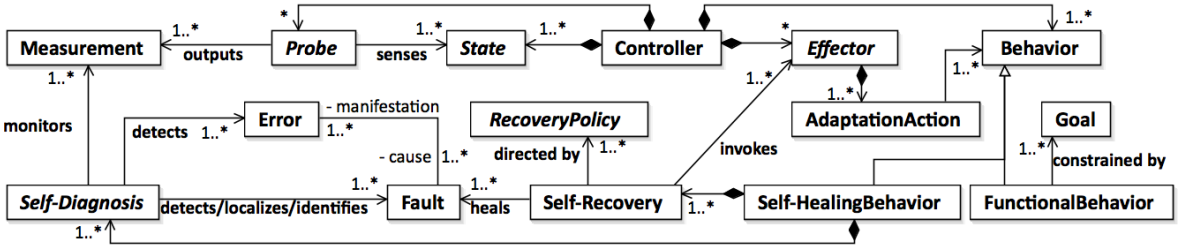


Figure 12 Concepts Relevant to Self-Healing Behavior

### 4.2.1 Self-Diagnosis

Figure 13 presents the key concepts related to *Self-Diagnosis*, aiming to detect *Fault*s from *Measurement*s. Its fault detection ability can be classified into three levels: *FaultDetection*, *FaultLocalization*, and *FaultIdentification* [19]. The diagnosis at the *FaultDetection* level can only detect the occurrence of faults; the *FaultLocalization* level diagnosis can determine which kind of faults has happened, and the diagnosis at the *FaultIdentification* level can further determine the severity of a fault. For instance, in the running example, the diagnosis of the servo fault can identify the magnitude of lift loss of a servo, and thus it belongs to the *FaultIdentification* level.

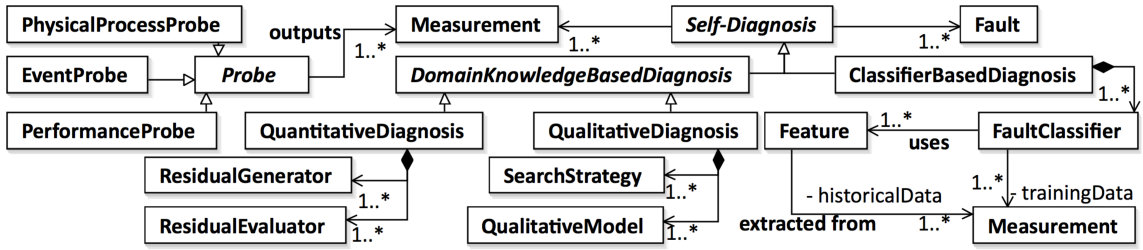


Figure 13 Concepts Relevant to Self-Diagnosis

A *Self-Diagnosis* behavior can either be realized based on prior domain knowledge (*DomainKnowledgeBasedDiagnosis*) or constructed by analyzing historical data (*ClassifierBased Diagnosis*). For the former, there are two viable options. One is

*QuantitativeDiagnosis*, in which domain knowledge is expressed as a quantitative model specifying mathematical relations between the input and output of a system [20]. Faults are detected by checking inconsistencies (residues) between the system's actual and expected output calculated from the model, via *ResidualGenerator* and *ResidualEvaluator*. The other option is *QualitativeDiagnosis*, which expresses domain knowledge in a *QualitativeModel*, i.e., qualitative relations between different system elements [20], such as cause-effect relations or fault trees. During execution, *SearchStrategies* are used to explore the *QualitativeModel* to detect faults.

For *ClassifierBasedDiagnosis*, it extracts *Feature*s from historical execution data (*Measurement*s) and mines relations between *Feature*s and different kinds of faulty states. Based on extracted relations, *FaultClassifier*s are constructed and used to classify system states as normal or faulty.

*Probe*s provide *Measurement*s required by *Self-Diagnosis* to detect faults. According to the type of *Measurement*s, *Probe*s are classified into *PerformanceProbe*s, *EventProbe*s, and *PhysicalProcessProbe*s. *PerformanceProbe*s are used to monitor a system's performance such as response time and throughput. *EventProbes* monitor *Controller*s' behaviors described as a trace of events such as function calls and exceptions. *PhysicalProcessProbe*s supervise the *State* of *PhysicalProcess*es. In the running example, all three kinds of *Probe*s are utilized, including an interface for monitoring the latency of radio control channel (*PerformanceProbe*), an interface for discovering unhealthy sensors (*EventProbe*), and an interface for obtaining the vehicle's position (*PhysicalProcessProbe*).

### 4.2.2 Self-Recovery

After fault diagnosis, a *Self-Recovery* behavior decides which *AdaptationAction*(s) to take to handle the detected fault, directed by *RecoveryPolicies*, as shown in Figure 14. *Effector*s provide *Self-Recovery* behaviors the ability to modify the system. According to the type of modification, *Effector*s can be classified into three types: *ParameterEffector*s for adjusting system components' parameters [21], *ArchitectureEffector*s for adding, removing, or replacing system components [22], and *ControlEffector*s for changing

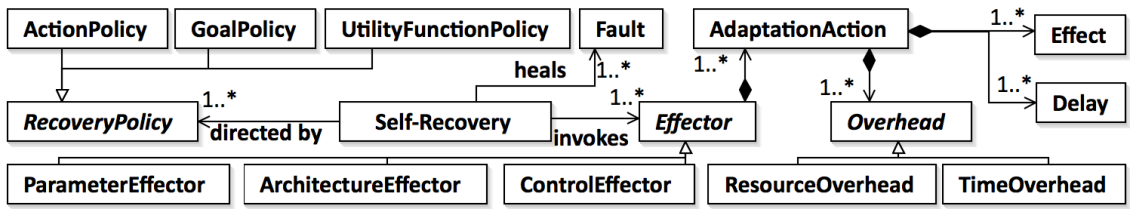


Figure 14 Concepts Relevant to Self-Recovery

*FunctionalBehavior*(s) in response to faulty conditions. Since the *Effect*s of *AdaptationAction*s are different and they may have different *Overhead*s and *Delay*s, a *Self-Recovery* behavior has to find an optimal action to recover a system from a detected fault. Because the effects, delays, and overheads are normally not observable from the outside of systems, they cannot be directly used for testing and therefore are not mandatory to be captured in test models.

In the literature, there are three types of *Recovery Policy* [23]: *ActionPolicy*, *GoalPolicy,* and *UtilityFunctionPolicy*. *ActionPolicy* can be seen as a pair in the form of <condition, action>. If the condition is satisfied, a corresponding action is executed [24], which is applied in the running example. *GoalPolicy* specifies desired states, which requires a sequence of modifications to be taken to make a system transit from a faulty state to a desired one [25]. *UtilityFunctionPolicy* defines an objective function to guide a system to desired states that have high utility values [26].

## 4.3 Uncertainty

During execution, *Uncertainties* may arise from interactions between SH-CPSs and their environments. Uncertainty is defined as the lack of knowledge of which value an *UncertainFeature* will take at a given point in time (*TimeInstance*) during testing [27] (Figure 15). For instance, a sensor's measurement is affected by noise, and we cannot determine the value of noise for a given point in time. Thus, the value of the noise is uncertain. *Universe* describes the set of all possible values that an *UncertainFeature* may take and each possible value is defined as a *Datum*. Taking the measurement noise as an example (Table 2), the *Universe* of the uncertain feature is an interval from -50 to 50, and every value within this interval is a *Datum*.

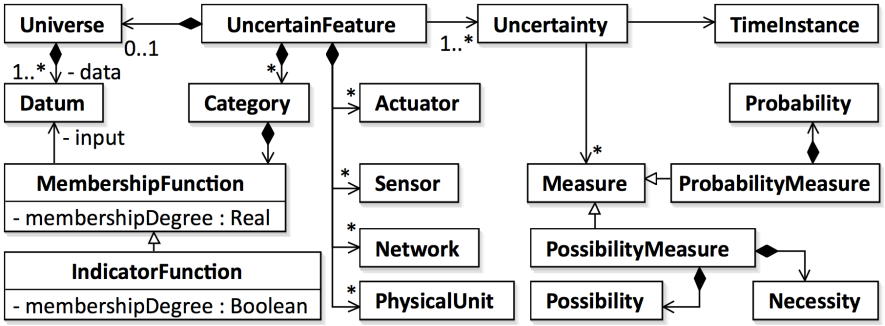

Figure 15 Concepts Relevant to Uncertainty

When we only have a qualitative knowledge, the set of possible values (*Datum*s) has to be described qualitatively. As shown in Table 2, the measurement noise of the GPS is described as low, medium and high, which are conceptualized as *Categorie*s in the conceptual model. A *Category* has one *MembershipFunction*, which determines to what extent a *Datum* belongs to the *Category*. More specifically, a membership function of a *Category* takes one *Datum* as input and outputs a real value between 0 and 1, representing the membership degree of the *Datum* to the *Category* [28]. A *Datum* could partially belong to multiple *Categories*, and in this case, each *Category* is a fuzzy set. *IndicatorFunction* is a specialized membership function, which only outputs 1 or 0, meaning that a *Datum* either belongs or does not belong to the *Category* associated with the *IndicatorFunction*. In this case, the corresponding *Category* is a crisp set. Table 2 shows an example of both cases. When using the *IndicatorFunction*, the boundaries of the three *Categories* (i.e., low, medium and high) are crisp, i.e., a noise value only belongs to one of the three *Categories*. In contrast, the boundaries defined by the *MembershipFunction* are fuzzy. In this case, for a noise value of 10, its membership degree is 0.5[5] to *Low*, 0.38 to *Medium*, and 0.12 to *High*.

Table 2 An Example of Uncertainty

| **Concepts** | **Example** |
|---|---|
| *UncertainFeature* | Measurement noise of the GPS |
| *Uncertainty* | Actual value of the noise at a given time instance |
| *Universe* | The interval from -50 to 50 |
| *Datum* | $\forall x, x \in [-50, 50]$ |
| *Category* | *Low, Medium, High* |
| *Membership Function* | $M_{Low}(x) = 1/(1 + e^{0.1\cdot(\lvert x\rvert-10)})$<br>$M_{Med}(x) = 1/(1 + e^{0.1\cdot(\lvert x\rvert-30)}) - 1/(1 + e^{0.1\cdot(\lvert x\rvert-10)})$<br>$M_{High}(x) = 1/(1 + e^{0.1\cdot(30-\lvert x\rvert)})$ |
| *IndicatorFunction* | $M_{Low}(x) = 1, if\ \lvert x\rvert \in [0,10);\ 0, others$<br>$M_{Medium}(x) = 1, if\ \lvert x\rvert \in [10,30);\ 0, others$<br>$M_{High}(x) = 1, if\ \lvert x\rvert \in [30,50];\ 0, others$ |

*Uncertainty* may be measured with different *Measure*s. From complete certainty to total ignorance, there exist five intermediate levels [27]. Table 3 shows these five levels along with their relations to *Measure*, *Datum*, and *Category*. For Level 1 Uncertainty, at a given point in time, the value of a feature is one value with a margin of error, i.e., one is certain that the value falls within this margin. For

Table 3 Uncertainty Levels

| **Level** | **Datum** | **Measure** | **Category** |
|---|---|---|---|
| Level 1 | A determined datum with a margin of error | N/A | N/A |
| Level 2 | A set of data | Probability | Related |
| Level 3 | A set of data | Possibility | Related |
| Level 4 | A set of data | N/A | Related |
| Level 5 | Known Unknowns | N/A | N/A |

[5] $M_{Low}(10) = 1/(1 + e^{0.1\cdot(|10|-10)}) = 0.5$

this reason, no qualitative specification (*Category*) or *Measure* is required for this level. In the running example, a servo's maximum thrust can be determined according to its product specification. However, this value is not accurate, and a tolerance interval is given to specify the range of possible values. Thus, the maximum thrust belongs to Level 1 Uncertainty.

Level 2 Uncertainty stands for the situation that a feature has alternate values with *Probabilities*. Thus, *ProbabilityMeasure* is used at this level to specify the *Probability* of a *Datum* or a *Category* to be true. For instance, the measurement noise of GPS conforms to a normal distribution. Through statistical analyses, the distribution can be determined, and thus it is a Level 2 Uncertainty.

For Level 3 Uncertainty, the probability of each possible value is unknown, but each possible value is bound with a ranked likelihood, which can be specified via the *Possibility* and *Necessity* of the *PossibilityMeasure*. Following the running example, due to the limited knowledge, the probability distribution of wind speed cannot be determined, and we can only compare the likelihoods of different potential values. Assume that the likelihood of medium wind speed is high and the ones of low and high speed are low. Their possibilities can be specified as 0.2, 0.7, and 0.2 to reflect their ranked likelihoods.

A Level 4 Uncertainty is the case when one can enumerate multiple alternative values of a feature but cannot rank their likelihoods, due to lack of knowledge, or disagreements among modelers [27]. Therefore, it is only possible to identify possible *Categories*, while the likelihood of each possible value is unknown. At last, Level 5 *Uncertainty* represents situations that the only thing known is that we do not know (i.e., *known unknowns*). Neither *Universe* nor *Measure* of an *Uncertainty* at this level is known. The only thing known is the existence of the uncertainty.

# 5 MoSH Modeling Framework

Based on the conceptual model (Section 4), UML profile of **M**odeling and **A**nalysis of **R**eal-**T**ime **E**mbedded Systems (MARTE) [7] and **U**ML **T**esting **P**rofile (UTP) [29], we implement a modeling framework — MoSH, which provides a set of UML stereotypes, datatypes and a modeling methodology to facilitate the specification of executable test models. To make the test model executable, we restrict the modeling notations to the executable subset of UML, which is defined by fUML [10]. Several metaclasses in the

subset are extended by the stereotypes from MoSH to specify self-healing behaviors and uncertainties. New data types are also provided to assist in setting the attributes of newly defined stereotypes. The stereotypes and datatypes are organized into four profiles, as shown in Figure 16. They are applied in four steps to create executable test models (Section 5.1 to Section 5.4).

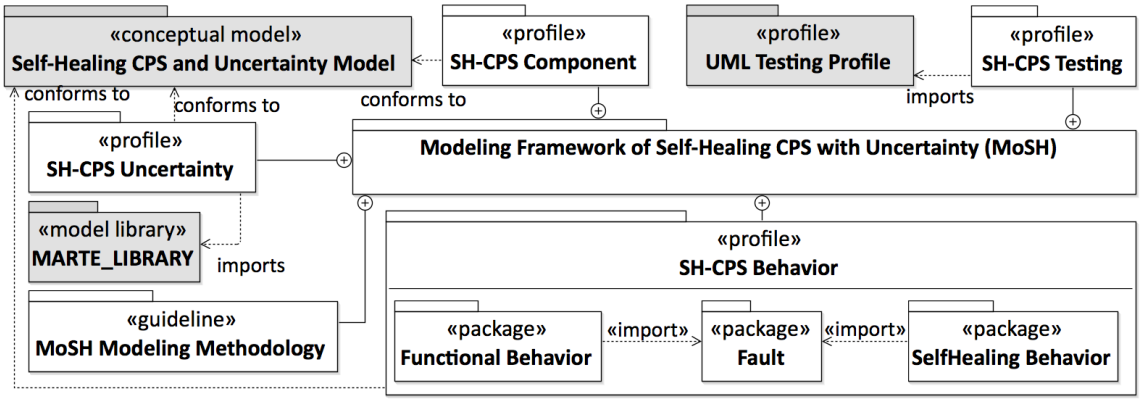


Figure 16 Overview of the MoSH Modeling Framework

## 5.1 Model System Structures with the SH-CPS Component Profile

This section presents the SH-CPS Component profile (Section 5.1.1) and methodology (Section 5.1.2).

### 5.1.1 SH-CPS Component Profile

Sensors, actuators, and controllers constitute the major components of a physical unit of an SH-CPS. Hence, six stereotypes are provided (Table 4) to annotate the role played by each component and to specify goals and uncertainties of the component, via the "goal" attribute of «PhysicalUnit»/«Controller» and "uncertainty" attribute of «PhysicalUnit»/«Sensor»/«Actuator»/ «Network». The "goal" is used as a test oracle, and the "uncertainty" specifies the uncertainty that needs to be introduced during testing.

Table 4 Stereotypes in SH-CPS Component Profile

| Stereotype | Metaclass | Attribute | Stereotype | Metaclass | Attribute |
|---|---|---|---|---|---|
| «SelfHealingCPS» | Package | type: ArchitectureType | «Sensor» | Behaviored Classifier | uncertainty: Uncertainty [*] |
| «Network» | Behaviored Classifier | uncertainty: Uncertainty [*] | «Actuator» | | uncertainty: Uncertainty [*] |
| «PhysicalUnit» | | goal: Constraint [*] uncertainty: Uncertainty [*] | «Controller» | | goal: Constraint [*] |

### 5.1.2 Model System Structure

First, physical units and networks are identified and specified as separate classes. Physical units can be further decomposed into sensors, actuators, and controllers. Figure 17 (A) shows a partial structural model of the running example, which consists of two physical units (*GroundControlStation*, *Drone*) connected through a network (*MAVLink*). *Drone* is decomposed into a *NavigationUnit*, a *GPS* and four *Servo*s.

All accessible state variables that can be queried via testing interfaces are specified as class attributes, such as the *mode* of the *NavigationUnit* and the *throttle* of the *Servo*. Operations capture the testing interfaces provided by corresponding components, including

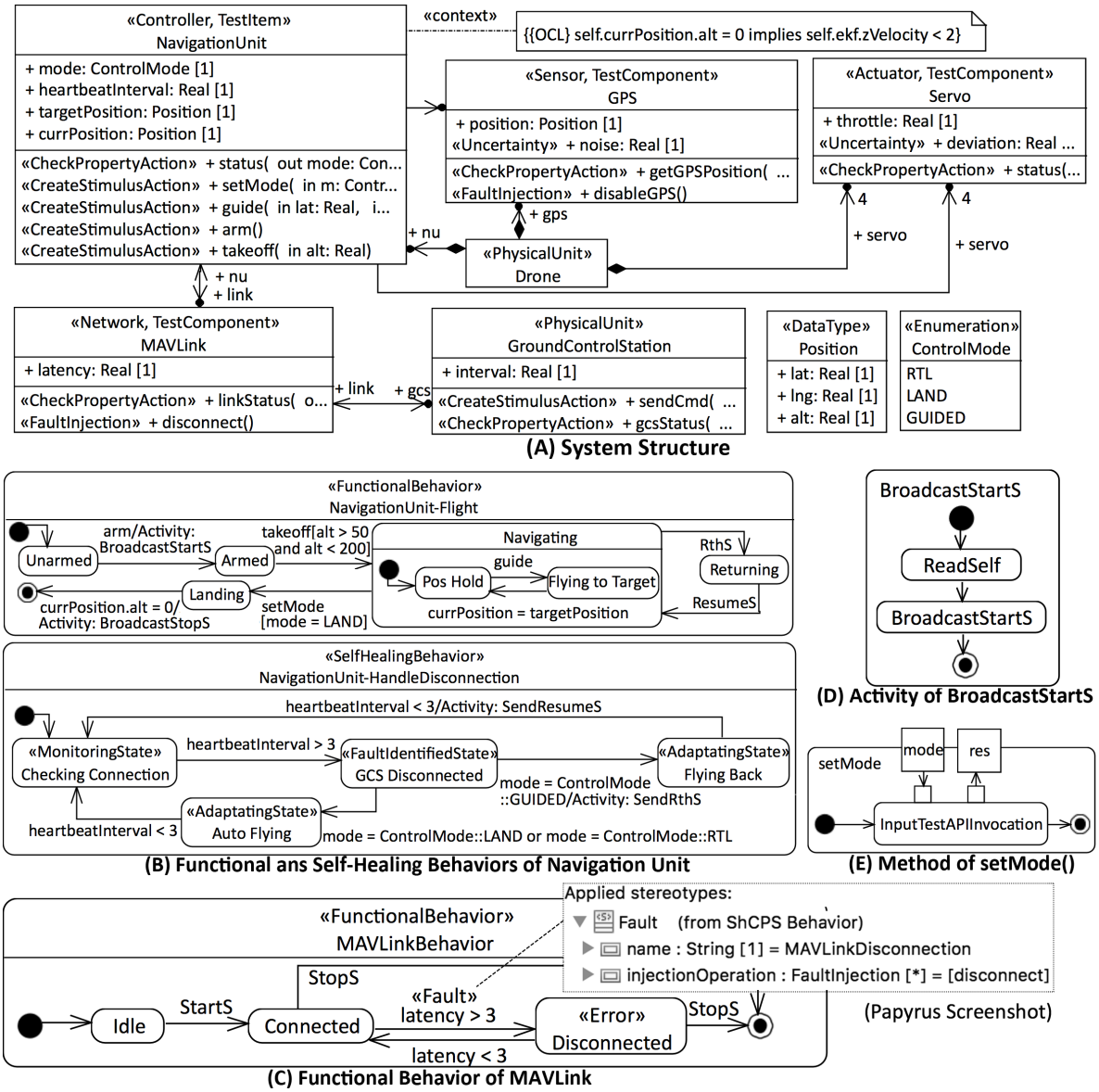


Figure 17 Executable Test Model of RAMA (Partial)

check property actions for querying state variables, create stimulus actions for manipulation, and fault injections for introducing faults to trigger self-healing behaviors. As shown in Figure 17 (A), every class has one or more operations for monitoring or controlling its corresponding component. Two fault injection operations, *disconnect()* and *disableGPS()*, are also implemented to simulate the fault of disconnection and loss of *GPS* signals.

Using the stereotypes' attributes, testers can specify the goals of physical units and controllers as OCL constraints. The constraints have to be defined on class attributes so that they can be validated based on the attributes' values obtained via testing interfaces. A goal of the *NavigationUnit* is shown in Figure 17 (A), which is to avoid a crash on the ground, i.e., when the *Drone* lands on the ground (*self. currPosition.alt = 0*), its vertical velocity should be below 2 meters per second (*self.ekf.zVelocity < 2*).

## 5.2 Model Behaviors with SH-CPS Behavior Profile

This section presents the SH-CPS Behavior profile (Section 5.2.1) and the guidelines to model functional behaviors, self-healing behaviors, and their interactions (Section 5.2.2 to Section 5.2.4).

### 5.2.1 SH-CPS Behavior Profile

This profile is proposed to specify expected self-healing behaviors. Since the objective of self-healing behaviors is to recover functional behaviors from faults, the expected functional behaviors should also be captured to assess the utilities of self-healing behaviors. As shown in Table 5, «FunctionalBehavior» is provided to annotate the behaviors, and to specify potential faults and consequent faulty states that may appear in a functional behavior via its "fault" and "error" attributes. A functional behavior is to be specified as a UML state machine to capture its normal and faulty states. «Fault», extending UML metaclass *ChangeEvent*, is provided to define a potential fault and specify a fault injection operation (via its "injectionOperation" attribute) to trigger the self-healing behavior that is to heal the fault. Its change expression defines the condition, under which a fault is considered occurred. As shown in Figure 17 (C), the disconnection of the *MAVLink* is a potential fault. It is specified as a *ChangeEvent* stereotyped with «Fault». Its change expression ("latency > 3") defines that the fault occurs if the latency exceeds 3 seconds and the *disconnect()* operation can be used to introduce this fault. When a fault occurs, i.e., a

fault is injected, the state of a functional behavior switches from a normal state to a faulty one stereotyped with «Error», which enables TM-Executor to directly identify the faulty state and recognize when a self-healing behavior needs to be performed.

Table 5 Stereotypes in the SH-CPS Behavior Profile

| Package | Stereotype | Metaclass | Attribute |
|---|---|---|---|
| Fault | «Fault» | ChangeEvent | name: String, injectionOperation: FaultInjection [*] |
| | «Error» | State | name: String |
| Functional Behavior | «FunctionalBehavior» | StateMachine, Region | fault: Fault [*], error: Error [*] |
| SelfHealing Behavior | «SelfHealingBehavior» | StateMachine, Region | fault: Fault [1..*] |
| | «MonitoringState» | State, StateMachine | measurement : Property [1..*] |
| | «FaultIdentifiedState» | State, StateMachine | fault: Fault [1..*] |
| | «AdaptatingState» | State, StateMachine | name: String |

To capture the logic of fault diagnosis and recovery, we choose to use UML state machines to model self-healing behaviors. «SelfHealingBehavior» is provided to annotate the behavior and specify which faults can be recovered by it via its "fault" attribute. Based on the stereotype attribute, TM-Executor can decide if a correct self-healing behavior is triggered by injecting a fault. If the behavior is not triggered, it means that the behavior failed to detect the fault and a potential implementation fault is found.

To test if a self-healing behavior can correctly detect a fault and apply proper actions to recover from the fault, testers need to specify one or more conditions, under which the fault is to be detected, and the recovery policy, which determines the selection of recovery actions. First, the logic of fault detection is specified via the transitions between «MonitoringState»s and «FaultIdentifiedState»s. «MonitoringState» is defined to annotate the states, where the self-healing behavior tries to detect faults based on measurements from probes. «FaultIdentifiedState» denotes that the self-healing behavior has detected, localized, or identified a fault. Second, the transitions between «FaultIdentifiedState»s and «AdaptatingState»s capture the recovery policy. «AdaptatingState» annotates the actions that are to be used by the self-healing behavior to recover from the fault. The following sections explain how to use these stereotypes to model functional and self-healing behaviors in details.

### 5.2.2 Model Functional Behaviors

For all identified classes (Section 5.1.2), testers need to specify their functional behaviors first, and then specify self-healing behaviors, i.e., define how to restore the functional behaviors to normal states in case of faults. The functional behaviors are modeled as UML state machines. Each state in the state machines should be precisely defined by state

invariants, i.e., OCL constraints on class attributes (Section 5.1.2). During test execution, any inconsistency between the system's actual state and the active state of the state machine indicates a potential implementation fault[6].

In state machines, a transition between two states models a valid fragment of behavior [30], which can be triggered by a *CallEvent*, *SignalEvent*, or *ChangeEvent*. *CallEvent*s represent invocations from external systems or users via operational calls such as the transition between the *Armed* and *Navigating* states in Figure 17 (B). Along with a *CallEvent*, a *Guard* (OCL constraint) can be specified to define valid ranges of inputs that can be used to invoke an operation. *SignalEvent*s capture interactions among different state machines. Via sending signals in effects or state activities, firing a transition in one state machine can lead to transitions in other state machines being triggered. *ChangeEvent*s are used to model internal changes such as the event "*currPosition == targetPosition*" shown in Figure 17 (B).

Based on test requirements, the faults that are to be healed by self-healing behaviors are specified as *ChangeEvent*s stereotyped with «Fault». The *ChangeEvent*s are used as triggers of transitions from normal states to error states to specify how functional behaviors are affected by the faults.

### 5.2.3 Model Self-Healing Behaviors

Self-healing behaviors are also modeled as UML state machines focusing on fault diagnosis and recovery. The logic of fault diagnosis is specified via the transitions between «MonitoringState»s and «FaultIdentifiedState»s. Triggers of the transitions are specified as *ChangeEvent*s, whose change expressions define criteria used to detect, localize or identify faults, such as the transition from "*Checking Connection*" state to "*GCS Disconnected*" state in Figure 17 (B), capturing the logic of fault detection for the disconnection fault.

For fault recovery, there are three kinds of recovery policies (Section 4.2.2). If the policy is action, goal or utility based and the goal or utility is defined on state variables that are accessible via testing interfaces, then the policy can be modeled as transitions from «FaultIdentifiedState»s to «AdaptatingState»s. The triggers of the transitions are specified as *ChangeEvent*s whose change expressions describe which recovery action is to be used.

[6]In case a system behaves differently from a test model that is derived from an incomplete requirement, it only indicates that a potential fault has been detected. Developers or designers who have more knowledge of the requirement can determine whether it is indeed an implementation fault.

As shown in Figure 17 (B), there are two ways to handle the disconnection fault. When the *NavigationUnit*'s mode is *LAND* or *RTL*, no manual control is required to control the flight, and the *Drone* keeps on its current task. Otherwise, the mode is changed from *GUIDED* to *RTL* under which the *Drone* flies back to its launch position. For a goal/utility based policy whose goal/utility is not defined on accessible state variables, testers have to identify invariants of the goal/utility, define them on accessible state variables, and specify them as class or state invariants. By checking them during testing, TM-Executor can determine if a wrong recovery action is performed.

The transition from an «AdaptatingState» to a «MonitoringState» is used to specify the behavior after the fault has been successfully healed. As shown in Figure 17 (B), as soon as the connection of *MAVLink* is rebuilt (i.e., *heartbeatInterval* < *3*), the flight mode is changed back to *GUIDED* to resume the flight.

### 5.2.4 Model Interaction Behaviors

Besides defining functional and self-healing behaviors in separate UML state machines, testers also need to specify interactions among them to facilitate the overall execution.

In UML state machines, four model elements can be targeted for specifying an interaction: the *Entry*, *Exit,* and *doActivity* of a state, and the *Effect* of a transition [11]. Their execution semantics are different, and testers can choose the one that suits the context best. An *Entry* is executed synchronously before activating a state. When the execution completes, the state's *doActivity* (if exists) is invoked asynchronously. When the state is to be exited and its *doActivity* is still running, the execution is aborted. An *Exit* behavior is executed synchronously before exiting a state. When a transition is triggered, its source state is exited first, and then its *Effect* is executed. As soon as the execution completes, the transition's target state is entered.

Interaction behaviors mainly involve sending *SignalEvent*s from one state machine to the others. According to the fUML standard [10], an interaction behavior can be specified as either an activity diagram or an opaque behavior with its method defined in the **A**ction **L**anguage for **f**UML (ALF) [31]. For instance, *BroadcastStartS* is an effect of a transition and defined as an activity diagram (Figure 17 (D)). Its execution semantics is to broadcast a given signal to all classes that have a direct association with the current class. When the transition is fired, the *MAVLink* will receive a *StartS* signal, triggering it to enter the *Connected* state.

## 5.3 Specify Uncertainties using SH-CPS Uncertainty Profile

We present the SH-CPS Uncertainty profile in Section 5.3.1 and the guideline to use it in Section 5.3.2.

### 5.3.1 SH-CPS Uncertainty Profile

For testing, uncertainty is the *lack of knowledge of which value an uncertain feature will take at a given point of time during testing*. As explained in Section 4.3, an uncertainty is specified by defining its universe, categories, and measure. Accordingly, «Uncertainty» is defined, along with "universe", "category", and "measure" attributes (Figure 18), each of which corresponds to a newly defined datatype.

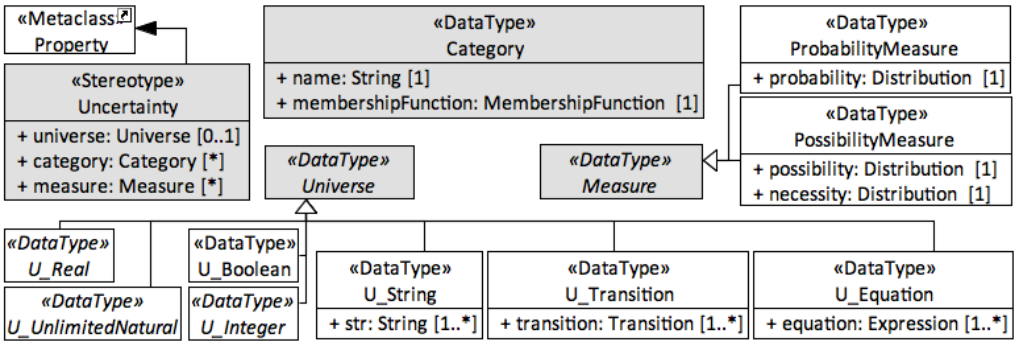


Figure 18 SH-CPS Uncertainty Profile

The *Universe* represents a collection of values. According to the type of value contained in a universe, we define seven subtypes: *U_Boolean*, *U_Integer*, *U_Real*, *U_UnlimitedNatural*, *U_Transition*, *U_String*, and *U_Equation*. For non-numerical datatypes (*U_Transition*, *U_String*, and *U_Equation*), the universe is specified as a collection of listed values. For numerical ones (*U_Integer*, *U_Real*, *U_UnlimitedNatural*), the universe can either be specified by enumeration or as an interval. *Category* is defined to specify a qualitative description of an «Uncertainty». As explained in Section 4.3, the elements of a *Category* are determined by a *MembershipFunction*. Based on the MATLAB fuzzy library [32], we provide six kinds of *MembershipFunction*[7]. *Measure* is defined to quantify uncertainty. According to testers' knowledge, *ProbabilityMeasure* or *PossibilityMeasure* can be used to specify the measurement of uncertainty; *ProbabilityMeasure* uses a probability distribution to quantify the chance of possible values, and *PossibilityMeasure* uses possibility and necessity distributions to map each

[7] Include Gauss, Generalized bell, Triangular, Difference between two sigmoid, Pi-shaped, Sigmoid functions.

possible value to a rankable likelihood. To specify the measurement of uncertainty, this profile imports six discrete and five continuous distributions [8] from the probability distribution library of MARTE [7].

### 5.3.2 Model Uncertainty

To test systems under uncertainties, testers need to define the uncertainties. For each specified class (Section 5.1.2), testers have to identify uncertain features according to testing specification and their domain knowledge. For instance, the measurement noise of GPS cannot be determined at a given point of time during execution, and thus the noise is identified as an uncertain feature, which is specified as a class attribute stereotyped with «Uncertainty». Via the "universe", "category", and "measure" attributes of «Uncertainty», the uncertainty associated with the uncertain feature can be quantified[9].

According to the type of an uncertain feature, a type of *Universe* is chosen to specify its valid range. For a numerical feature, if its value varies within a range, an interval datatype (*U_IntegerInterval*, *U_RealInterval*, *U_UnlimitedNaturalInterval*) should be assigned. Otherwise, a vector can be used to list all possible values. Depending on the level of uncertainty (Section 4.3), the uncertainty's category and measure are specified differently. For a Level 1 uncertainty, at a given point in time, the value of the uncertain feature is determined with a margin of error. It is specified via "universe" attribute of «Uncertainty». For Level 2 and Level 3 uncertainties, before quantifying the likelihood of each value, testers should decide if their knowledge about the uncertain feature is qualitative or quantitative. If it is qualitative, a membership function is used to define a category for each descriptive term, such as the three categories: *Low*, *Medium*, and *High*, defined for the measurement noise of GPS in Table 2.

After defining categories, testers need to specify the measurement of each uncertainty. For Level 2 uncertainty, since the probability of each value is known, a probability measure is used to describe the likelihood. Possibility measure is adopted for Level 3 uncertainty to state the rankable likelihood of each value. For both levels of uncertainties,

[8]Include Poisson, Bernoulli, Categorical, Logarithmic, Discrete Uniform, Exponential, Gamma, Normal, Triangular, and Trapezoidal distributions.

[9] If the value of an uncertain feature depends on the values of some other uncertain features, OCL constraints can be used to specify the dependency.

appropriate probability, possibility, or necessity distributions can be chosen from the set of distributions provided by the uncertainty profile (Section 5.3.1).

Since measures of Level 4 and Level 5 uncertainties are unknown, they cannot be explicitly specified in the model. However, as testing proceeds, more knowledge about such uncertainties may be obtained, which make them transform to Level 3, Level 2 or even Level 1 uncertainties. Afterward, they can be precisely specified and used in the testing approach.

## 5.4 Model Testing Utilities with SH-CPS Testing Profile

We present the SH-CPS Testing profile in Section 5.4.1 and guidelines to specify test configurations and testing interfaces that are required to execute a test model together with a system (Section 5.4.2).

### 5.4.1 SH-CPS Testing Profile

Figure 19 presents the SH-CPS Testing Profile. It imports five stereotypes from the Test Architecture and Test Behavior packages of UTP v2 [29], as these stereotypes are standardized and have been precisely defined by UTP v2. «TestItem» denotes the testing target, i.e., controllers of SH-CPSs. The class stereotyped with «TestItem» signifies the entry point of test execution. «CreateStimulusAction» and «CheckPropertyAction» extend *BehavioralFeature* representing testing interfaces used for controlling and monitoring systems. «CheckPropertyAction» annotates operations that are used to query the values of state variables. «CreateStimulusAction» should be applied on operations that are used to control the behaviors of a system. «FaultInjection» is a specialized «CreateStimulusAction» for faults injections. «TestComponent» represents sensors, actuators, networks, and external systems, which are simulated/emulated. Configurations of simulators/emulators can be specified via "configuration" attribute of «TestComponent».

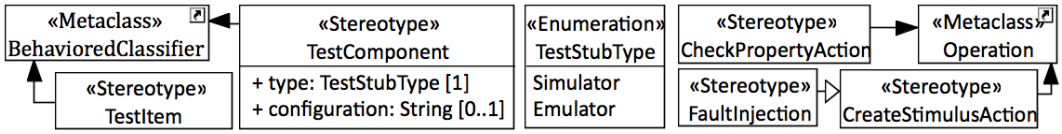


Figure 19 SH-CPS Testing Profile

### 5.4.2 Model Test Utilities

The final step of the modeling is to specify test configurations and bind the testing interfaces with the defined operations to achieve an executable test model. First, the main

class, which contains the entry point of test execution, is stereotyped as «TestItem». Second, sensors, actuators, and physical processes, which are to be simulated/emulated, are annotated with «TestComponent». The "configuration" attribute of «TestComponent» is used to specify the configuration parameters of simulators/emulators. Third, operations defined in class diagrams need to be stereotyped with «CreateStimulusAction», «CheckPropertyAction» or «FaultInjection» to distinguish their functionalities. For «CreateStimulusAction» and «FaultInjection», their input parameters capture the input of corresponding testing interfaces, and they can optionally have an output parameter to represent the result of an invocation, i.e., success or failure. For «CheckPropertyAction», it has no input parameters, and its output parameters declare the state variables that can be queried via the «CheckPropertyAction».

The method of «CreateStimulusAction», «CheckPropertyAction» or «FaultInjection» can be specified using either an activity diagram or an opaque behavior with its method defined in ALF [31]. Figure 17 (E) presents the method of a «CreateStimulusAction» — *setMode()*. This operation is to set the *NavigationUnit*'s control mode and returns *res* to signify the result. In the method, an *OpaqueAction — InputTestAPIInvocation*, is defined to bind the operation with a corresponding testing interface, whose Uniform Resource Identifier (URI) is specified in the body of the action.

# 6 TM-Executor Framework

TM-Executor was implemented based on existing standards of fUML [10], PSSM [11] and FMI [33]. It takes charge of (1) executing and testing a system against a test model, (2) simulating the effect of uncertainties, and (3) orchestrating the execution of test model, system, and simulators/emulators. This section first gives an overview of TM-Executor and then explains these functionalities in sequence.

## 6.1 Overview

Figure 20 presents an overview of TM-Executor and its relations with MoSH. The MoSH modeling framework is implemented in Papyrus, a UML modeling environment [34]. With MoSH, testers can create an executable test model, which is taken by TM-Executor as input to perform uncertainty-aware executable model-based testing. In TM-Executor, the test model is executed by an *ETM Execution Engine*, which is built on Moka [35] — a

UML model execution platform that realizes the standards of fUML [10] and PSSM [11]. During execution, the execution engine periodically obtains the values of state variables from the system under test, and a *Constraint Checker* uses the values to evaluate the goals of classes (Section 5.1.2) and the invariants of current active states (Section 5.2.2). If any constraint is evaluated to be false, it means the system fails to behave consistently with the test model.

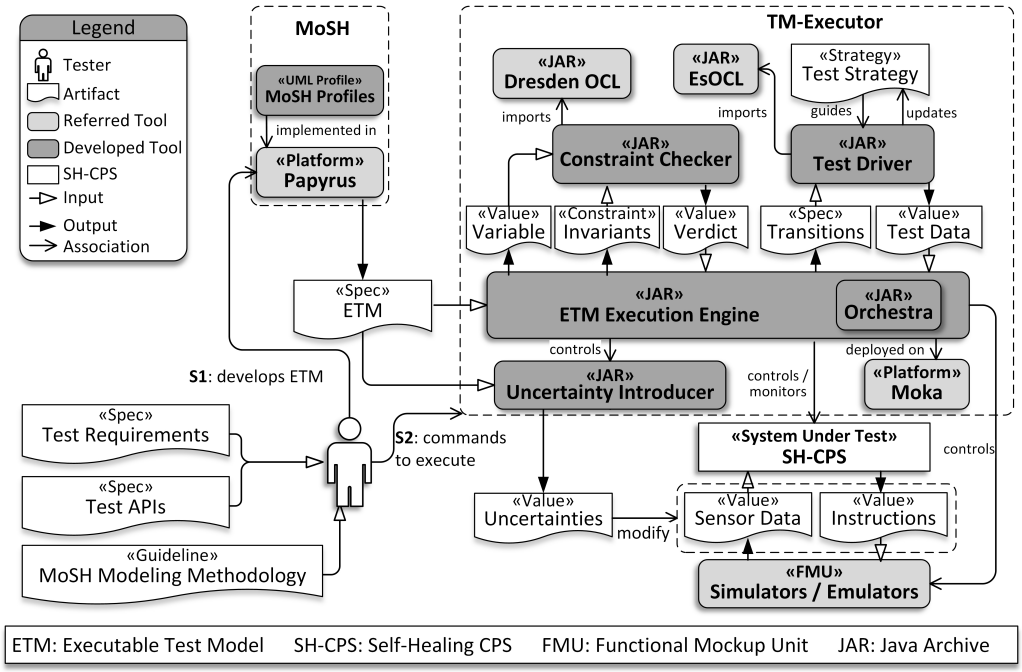


Figure 20 *TM-Executor* Framework

To drive the execution of the test model and the system under test, a *Test Driver* is implemented to generate testing stimuli during execution. Whenever the test model reaches a stable state configuration, in which the active states of the model are not changing, the *Test Driver* uses a testing strategy to select an outgoing transition of current active states, parses its trigger and guard, and uses EsOCL [36] — an OCL constraint solver, to generate a valid input for firing the transition.

Besides operation invocations, uncertainties need to be introduced in the testing environment to test a system under uncertainties. When the system interacts with the environment via sensors or actuators, an *Uncertainty Introducer* generates a value for each specified uncertainty and introduces it into sensor data or actuator instructions to simulate the effect of the environmental uncertainty.

Since the software of the SH-CPSs is to be tested with simulators/emulators, the test model, system, and simulators/emulators have to be executed coordinately to fulfill the

executable model-based testing. Typically, these three kinds of executable objects are implemented by distinct modeling/programming languages and from the perspectives of diverse domains. For instance, the test model, modeled by state machines, exhibits discrete behaviors. On the contrary, a simulator, modeled by differential equations, presents continuous semantics. To orchestrate the executions of the diverse objects, an *Orchestra* is implemented based on the **F**unctional **M**ock-up **I**nterface (FMI) standard [33]. It takes charge of propagating values through the executable objects and synchronizing their executions.

## 6.2 Executable Model-Based Testing

The *ETM Execution Engine* executes a test model according to an extended version of fUML [10] and PSSM [11]. The execution process and the extensions are presented in the following subsections.

### 6.2.1 Execution Process

When a test model is executed, the *ETM Execution Engine* first instantiates all class objects (directly or indirectly) associated with the main class stereotyped with «TestItem» (Section 5.4.2). Then, the engine starts the classifier behavior of each object, i.e., executing UML state machines (Section 5.2). When all behaviors have been started, the engine registers all events that can trigger the outgoing transitions of current active states. Next, the engine synchronizes the execution of the model with the executions of the system under test and simulators/emulators, to coordinate their executions.

After synchronization, the engine automatically invokes the check property actions specified in the test model (Section 5.4.2) to update values of state variables. With the updated values, the *Constraint Checker* verifies the invariant of current active states and the goals of each class. If any constraint is violated, a potential fault is detected and the execution terminates. Otherwise, the engine evaluates the *ChangeExpression*s of registered *ChangeEvent*s and sends those whose *ChangeExpression*s are true. The sent events are matched with outgoing transitions of current active states. Each matched transition is traversed via three steps. First, the source state of the transition is exited. If its *doActivity* behavior is still running, the execution is aborted. If the source state has an *Exit* behavior, it is executed before exiting the state. Second, the transition is traversed, and the *Effect* of the transition, if exists, is synchronously executed. Third, the target state of the triggered

transition is entered. If it contains *Entry* and *doActivity* behaviors, the *Entry* behavior is synchronously executed first, and then the *doActivity* is asynchronously executed. If any *Signal* is sent by the *Effect*, *Entry*, *doActivity*, or *Exit* behavior, a corresponding *SignalEvent* is generated and triggers other transitions.

When the active states of all state machines are not changing, the *Test Driver* invokes an operation to trigger an outgoing transition that is selected by a testing strategy. Afterward, the engine executes the method of the operation to call the corresponding testing interface. Meanwhile, a *CallEvent* is generated to trigger the outgoing transition. In this way, the test model and the system are being stimulated by the same invocation. The execution continues until all state machines reaching a final state. Appendix A presents an activity diagram of the execution process.

### 6.2.2 Extensions to fUML and PSSM

To facilitate test execution, TM-Executor extends fUML and PSSM in four ways. First, TM-Executor extends fUML to execute the test model specified with MoSH. For instance, operations stereotyped with «CreateStimulusAction» and «CheckPropertyAction» have specific semantics in our case, and thus we extended the existing semantics of the *Operation* defined in fUML. Second, for certain UML model elements (i.e., *BroadcastSignalAction* and *ChangeEvent*), neither fUML nor PSSM defines their execution semantics. Thus, we defined their semantics and implemented them in the execution engine. Third, there exist defined semantics in PSSM for certain UML metaclasses, but they do not sufficiently serve our purpose. Thus, we extended them. For instance, the execution semantics of *State* and *Transition* were extended to consider state invariants. Fourth, we newly defined three types of *OpaqueAction* to facilitate the specification of testing interfaces. The detailed implementations are presented in the corresponding technical report [18], and Appendix B summarizes the extensions.

## 6.3 Introduce Uncertainties

In TM-Executor, environment changes are controlled by simulators/emulators. When a system interacts with an environment via sensors or actuators, the data measured by sensors or the actuations performed by actuators are potentially affected by uncertainties. The uncertainties, captured in a test model, have to be introduced when the interaction happens. The *Uncertainty Introducer* fulfills the task. It generates uncertainty values in

different ways for the three levels of uncertainties (Section 5.3.2). The values are used to modify data generated by sensors for measurement uncertainties, alter instructions sent to actuators for actuation deviations, and detain delivery of sensor data and actuation instructions for network latency.

For level 1 uncertainties, their possible uncertainty values are defined as an interval, representing a determined value with a margin of error. By selecting a value from the interval, the value of a level 1 uncertainty is generated. For level 2 and level 3 uncertainties, the knowledge of uncertainties can be quantitative or qualitative. For the qualitative knowledge, several categories are defined for the uncertainty values (Section 5.3.2), and probability or possibility distributions define the likelihood of each category. In this case, the *Uncertainty Introducer* first selects a category based on the probability or possibility distribution, and then it generates an uncertainty value based on the membership function of the category. If the knowledge is quantitative, probability or possibility distribution is specified to measure the uncertainty. For a probability distribution, *Uncertainty Introducer* directly generates uncertainty values according to the distribution. For a possibility distribution, it is first transformed into an equivalent probability distribution [37], which is used to generate an uncertainty value.

## 6.4 Orchestrate Execution

Based on the FMI standard[10] [33], we implemented a co-execution algorithm in the *ETM Execution Engine* to coordinate the execution of the test model, system, and simulators/emulators. The following two subsections explain the FMI based co-execution architecture and the algorithm respectively.

### 6.4.1 FMI Based Co-Execution

FMI is a standard to support co-execution of hybrid models [33] and is particularly suitable for SH-CPSs, as they often contain subsystems designed with diverse modeling paradigms [38]. In the architecture of FMI based co-execution (Figure 21), every executable model is wrapped as a **F**unctional **M**ock-up **U**nit (FMU). Each FMU can be implemented by its

[10]Though there are several other co-simulation standards, such as HLA (coming from military applications), SMP (in the space domain), FMI gained the most attention from both research and industry. Thus, it is used in our work.

simulation language, but its interfaces[11] and its I/O dependence relations are defined by the FMI standard. Via the standard interfaces and based on the I/O dependence relations, a *Master* program orchestrates communications and executions of FMUs.

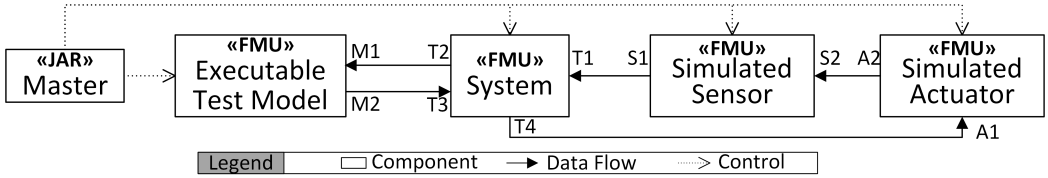


Figure 21 Architecture of the FMI Based Co-Execution

In TM-Executor, the executable test model, system, and simulators/emulators are all executed as FMUs (Figure 21). Their standard interfaces are either directly implemented in their dynamic models or indirectly realized by an FMI wrapper [33]. The *ETM Execution Engine* plays the role of the *Master* program. It uses the standard interfaces of the FMUs to orchestrate the co-execution. Because the FMI standard does not provide the algorithm used by *Master* for co-execution, inspired by Broman et al. [38], we implemented a co-execution algorithm, which is presented in the next section.

### 6.4.2 Co-Execution Algorithm

Our co-execution algorithm (Algorithm 1) takes three inputs. The first is a set of interconnected FMUs, similar to the one shown in Figure 21. The connections between the ports of the FMUs reflect their I/O dependencies. For instance, input port *T1* depends on output port *S1*, and input port *M1* depends on output port *T2*. Note that although there is a loop in Figure 21, port *S1* does not depend on port *S2*. Thus, there is no cyclic dependency. If there are cyclic dependencies among sensors or actuators, the Newton Raphson method [39] has to be used to compute the interdependent port values iteratively. As the FMI standard does not support the cyclic dependencies [33], we only focus on acyclic cases in this paper.

The second input of the algorithm is a set of ordered I/O dependencies. Because of I/O dependencies among ports, values of input and output ports have to be propagated in a right order to assure a determinate and correct co-execution. As the I/O dependencies form a directed graph (Figure 21) and the graph is acyclic, we can obtain a right propagation order by following the graph [38].

[11]There are four kinds of standard interfaces defined in FMI: **init**, which initializes the execution time of a FMU; **set**, which assigns a given value to a variable in a FMU; **get**, which queries the value of a variable in a FMU; and **doStep**, which performs an execution step on a FMU, using a given step size Δt.

***Algorithm 1*** ***Coexecute(List<FMU> FMUs, List<Dependence> orderedDeps, StepSize h)***:

***Input*** ***FMUs*** *is the set of FMUs that to be executed.*

***orderedDeps*** *are the ordered I/O dependencies among the FMUs.*

***h*** *is the maximum co-execution step size, acceptable for all the FMUs.*

***Output*** ***stepNum*** *is the total number of steps performed in the co-execution*

***Begin***

```
1  for each M in FMUs
2     M.init()
3  stepNum ← 0
4  while FMUs not terminate
5     for each relation (Msrc.src → Mtar.tar) in orderedDeps
6        Vsrc ← get(Msrc, src)    //get the independent value from the source FMU
7        set(Mtar, tar, Vsrc)     // set the dependent port value for the target FMU
8     for each M in FMUs
9        doStep(M, h)
10    stepNum ← stepNum + 1
11 return stepNum
```

***End***

The third input of the algorithm is the step size, which is to be used in each execution step. Limited by its implementation, an FMU cannot perform an execution step with an arbitrary step size. For example, assume that a numerical integration is used in an FMU. The range of the integration restricts the maximum step size that FMU can perform. Therefore, before the co-execution, testers have to determine the maximum step size that is acceptable for all FMUs.

Taking the three inputs, the algorithm first initializes the time of each FMU to zero, i.e., the beginning time of the co-execution (L1, L2). After initializing the step number (L3), the algorithm continuously performs two-phase execution steps until termination (L4 ~ L10). In the first phase, the algorithm propagates values of input and output ports, according to the given I/O dependencies order (L5 ~ L7). Following that, it advances the execution of each FMU by a given step size *h* (L8, L9) and then updates the step number (L10). At last, the algorithm returns the number of steps that have been performed in the co-execution. Since the time semantics of UML state machines is discrete events, advancing the execution time of the test model only gives the model an opportunity to process received events and fire transitions. This corresponds to the synchronization mentioned in Section 6.2.1. Physical parts of SH-CPSs are simulated or emulated by simulators/emulators, whose execution semantics are continuous. Advancing the execution time of simulators/emulators means that they perform a simulation computation with a constant step size.

The time complexity of Algorithm 1 is O((N+2P)*S), where N is the number of FMUs that are used in the co-execution, P is the total number of ports of all FMUs, and S is the average number of steps performed in each co-execution. The space complexity of Algorithm 1 is O(N), i.e., the space complexity is linear with the number of FMUs.

# 7 Evaluation

Section 7.1 presents the experiment design, followed by the experiment execution (Section 7.2.2), and results (Section 7.3). Section 7.4 gives an overall discussion, and Section 7.5 presents threats to validity.

## 7.1 Experiment Design

The experiment was designed to answer three research questions (RQ1-RQ3), as shown in Table 6.

Table 6 Experiment Design

| RQ | Task | Metrics | Systems | Statistical Test |
|---|---|---|---|---|
| **1** | $T_1$: Mapping concepts and their relationships from SH-CPSs to the ones in CMSU | *PerCov*, *PerCorr* | VCS, TMS, RFID-SC, DSRL, ISR, APR, RAMA, PeMS, VSS | N/A |
| **2** | $T_2$: Creating executable test models with MoSH | *FunBeh, HealBeh, Diagnosis, Recovery, Uncertainty, TotalElem, PerStereo* | RAMA, PeMS, VSS | |
| **3** | $T_3$: Applying TM-Executor to test an SH-CPS with a random strategy. | *TraTime, ExitStateTime, EnterStateTime, ExeOpTime, GenDataTime, EvalTime, GenUncTime, DetectedFault, OpProcTime* | RAMA | Kruskal–Wallis test |

**RQ1**: Can CMSU cover all relevant concepts and their relationships identified in selected SH-CPSs?

With this RQ, we aim to assess if any concepts or relationships cannot be correctly mapped to the conceptual model. Doing so helps us to find missing concepts and incorrect relationships in CMSU. Note that we do not prove the completeness and correctness of CMSU. Instead, we report empirical data to increase the confidence that CMSU is complete and correct.

We, therefore, defined task $T_1$ to answer RQ1. In the task, nine SH-CPSs were used to assess the quality of CMSU: **V**ideo **C**onferencing **S**ystem (VCS) [40], **T**raffic **M**onitoring **S**ystem (TMS) [41], **R**adio-**F**requency **ID**entification **S**upply **C**hain (RFID-SC) [42], **D**istributed **S**ystems **R**esearch **L**ab (DSRL) [43], **I**ntelligent **S**ervice **R**obot (ISR) [44], **A**utomatic **P**ower **R**estoration **S**ystem (APRS) [45], RAMA [13], freeway **Pe**rformance

**M**easurement **S**ystem (PeMS) [46], and **V**ideo **S**treaming **S**ystem (VSS) [47]. Based on the specification of the nine systems, we first identified their main components. For each component, we captured its behaviors and the environmental uncertainties that may affect these behaviors. For self-healing behaviors, we further identified their strategies to detect and recover from faults. The identified components, behaviors and uncertainties were manually mapped to the concepts and relationships in CMSU. Based on the mapping, we calculated the **Per**centage of **Cov**ered Concepts and Relationships (*PerCov*) and the **Per**centage of **Corr**ectly Covered elements (*PerCorr*).

**RQ2**: Does MoSH provide a cost-effective way of creating executable test models?
With this RQ, we want to assess 1) the additional modeling effort that is required to apply MoSH to create executable test models, as compared with applying standard UML notations, and 2) the applicability of MoSH to model self-healing behaviors and uncertainties.

For this RQ, we defined task $T_2$, which applies MoSH to build executable test models for SH-CPSs. To measure the extra modeling effort required for applying MoSH, we calculated the **Total** number of model **Elem**ent (*TotalElem*) and the **Per**centage of **Stereo**typed model elements (*PerStereo*). For the applicability, we checked the numbers of **Fun**ctional **Beh**aviors (*FunBeh*), Self-**Heal**ing **Beh**aviors (*HealBeh*), Self-**Diagnosis** behaviors (*Diagnosis*), Self-**Recovery** behaviors (*Recovery*), and **Uncertainties** (*Uncertainty*) that can be specified with MoSH. For RQ2, we used RAMA [13], PeMS [46], and VSS [47] as we do not have sufficient specifications for the other six systems (for RQ1).

**RQ3**: How is the performance of TM-Executor regarding test execution?
With this RQ, we are interested in assessing how much time is required by TM-Executor to execute a test model, generate test data, evaluate constraints, and introduce uncertainties. For RQ3, we defined task $T_3$, which applies TM-Executor to test the RAMA system against the executable test model created in task $T_2$[12] under environmental uncertainties. For the evaluation purpose, we implemented a random testing strategy, which randomly selects an outgoing transition to generate a testing stimulus. To account for randomness, we conducted 100 runs for the experiment. In each run, TM-Executor executed RAMA

[12] We could not use PeMS and VSS to answer RQ3 as we didn't have access to their implementations.

together with the test model 1000 times, directed by the random testing strategy. The 1000 times of executions allow TM-Executor to cover most transitions specified in the test model.

We evaluated the performance of TM-Executor from four aspects. First, we measured the time cost of TM-Executor for each execution step of the test model, i.e., **Exit**ing a **State** (*ExitStateTime*), **Tra**versing a transition (*TraTime*), **Enter**ing a **State** (*EnterStateTime*), and **Exe**cuting an **Op**eration (*ExeOpTime*).

Second, we measured the time taken by TM-Executor to **Gen**erate test **Data** (*GenDataTime*). Because test data was obtained by solving guards specified in a test model, the *GenDataTime* was affected by the complexity of the guards, i.e., the complexity of OCL constraints on input parameters. Thus, we further assessed the effect of guard complexity on the *GenDataTime*. In total, there are 52 guards in the test model. According to three complexity metrics proposed in [48], we put the guards with the same complexity in one group and obtained four groups of guards (Table 7). We measured the *GenDataTime* for each guard covered by the 100 runs of the experiment. On average, each guard is covered 29125 times. We collected 196304, 175101, 371674, and 771395 samples of the *GenDataTime* for the four groups of guards, respectively. To check the normality of the *GenDataTime*s, we conducted the Shapiro–Wilk test for each group. The p-values of the tests are lower than $2 \times 10^{-16}$. Thus, the samples of the four groups depart from normality. Hence, we applied the Kruskal–Wallis test to check if the *GenDataTime*s of the four groups are significantly different.

Table 7 Descriptive Statistics of Guard and Invariant Groups

| Groups | #Types | Types | #Clauses | Group Size |
|---|---|---|---|---|
| **Guard Group 1** | 1 | Enumeration | 1 | 6 |
| **Guard Group 2** | | Real | 1 | 6 |
| **Guard Group 3** | | | 2 | 13 |
| **Guard Group 4** | | | 6 | 27 |
| **Invariant Group 1** | 2 | Enumeration & Real | 2 | 12 |
| **Invariant Group 2** | | | 3 | 6 |
| **Invariant Group 3** | | | 7 | 27 |
| **Invariant Group 4** | 1 | Enumeration | 1 | 13 |
| **Invariant Group 5** | | Real | 1 | 29 |
| **Invariant Group 6** | | | 2 | 12 |

*#Types* is the number of parameter types contained in a constraint.
*Types* are the parameter types contained in a constraint.
*#Clauses* is the number of clauses in a constraint.

Third, we measured the time cost of TM-Executor to **Eval**uate a class or state invariant (*EvalTime*), which is also defined as an OCL constraint. As the same for the *GenDataTime*, the *EvalTime* was also affected by the complexity of the invariant. The test model contains 99 invariants in total. We put the invariants with the same values of complexity metrics in one group and obtained six groups (Table 7). We measured the *EvalTime* whenever an

invariant is evaluated during the experiment. In total, we collected 650310, 300516, 1350723, 650501, 1449653, and 600632 samples of the *EvalTime* for the six groups of invariants. Since the p-values of the Shapiro–Wilk tests for the six groups of samples are lower than 0.05, the samples are not normally distributed. Thus, we conducted the Kruskal–Wallis test to check if the *EvalTime*s of the six groups are significantly different.

Fourth, we evaluated the time spent by TM-Executor to **Gen**erate an **Unc**ertainty value (*GenUncTime*). As explained in Section 6.3, an uncertainty value is generated based on universes, probability distributions, possibility distributions, or membership functions. To assess the effect of different ways of generating uncertainty values on the *GenUncTime*, we grouped uncertainties according to the methods of generating their uncertainty values. In total, there are ten uncertainties specified in the test model. One of them is Level 1 uncertainty, whose values are generated based on its universe. Seven of them are Level 2 uncertainty, whose values are generated according to their probability distributions. One is Level 3 uncertainty, whose values are generated based on its possibility distribution. The last one is also a Level 3 uncertainty, while its values are generated based on its possibility distribution and membership functions, as knowledge about this uncertainty is imprecise. We measured the *GenUncTime*, whenever TM-Executor introduces an uncertainty during the experiment. We collected 7445059, 52115413, 7445059 and 7445059 samples of the *GenUncTime* for the four groups of uncertainties. The p-values of the Shapiro–Wilk tests for the samples are lower than 0.05, and thus the samples depart from normality. Therefore, we applied the Kruskal–Wallis test to check if the *GenUncTime*s of the four groups are significantly different.

## 7.2 Experiment Execution

This section presents the execution processes for the three tasks defined in Section 7.1.

### 7.2.1 Identifying and Mapping Concepts ($T_1$)

Based on the specification of the nine SH-CPSs, we evaluated and improved CMSU's quality, following the steps summarized in Figure 22. Initially, we derived the conceptual model (CMSU V.1) from the literature (Activity A1 in Figure 22). To evaluate its quality, we identified SH-CPS related concepts as well as their relationships (Cons. & Rels. from CSs. V.1), from the nine SH-CPSs' specifications (Activity A2.1). Cons. & Rels. from CSs. V.1 contain necessary entities required to specify self-healing behaviors and uncertainties

of an SH-CPS. For each identified concept or relationship, we tried to manually find a counterpart in CMSU V.1 (Activity A2.2). If the counterpart is missing, we further investigated if the extracted concept or relationship is correctly identified. In case that it was correct, CMSU V.1 was revised to cover the missing concept. Otherwise, the incorrectly identified concept or relationship was fixed. After A2.2, we created a new version of the extracted concepts and relationships, i.e., Cons. & Rels. from CSs. V.2. At last, the refined conceptual model (CMSU V.2) was further refined by A3 via a mapping from Cons. & Rels. from CSs. V.2 to CMSU V.2. The final obtained CMSU V.3 is presented in Section 4 and was implemented as UML profiles (Section 4.1.2).

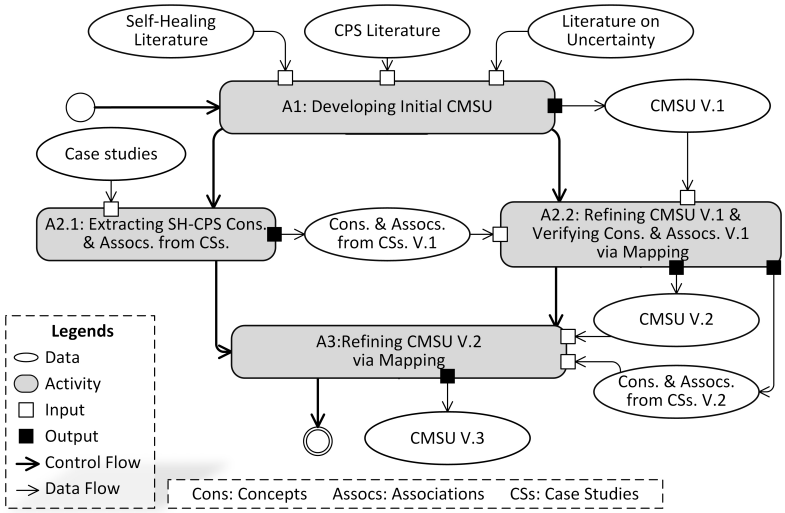


Figure 22 The Process of Developing CMSU

### 7.2.2 Modeling Executable Test Models with MoSH ($T_2$)

To answer RQ2, the first author of this paper applied MoSH to specify executable test models for RAMA [13], PeMS [46], and VSS [47], following the modeling methodology presented in Section 4.1.2. First, the architecture of each system was specified as a UML class diagram, with SH-CPS Component Profile applied. Second, the behaviors of each class were captured as UML state machines, and SH-CPS Behavior Profile was applied to model the potential faults and the logic of fault diagnosis and recovery. Third, SH-CPS Uncertainty Profile was applied to define environmental uncertainties that may impact the captured behaviors. To correctly define uncertainties, we used detailed product specifications of sensors and actuators, provided by manufacturers[13]. The specifications

[13] One example of the product specifications can be downloaded from: https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

specify the sensors' and actuators' characteristics, such as sensitivity, nonlinearity, and noise level. Based on the information, we identified and modeled the uncertainties in the test models. Fourth, UML activity diagrams were specified to bind testing interfaces with operations defined in each class. Meanwhile, SH-CPS Testing profile was applied to the class diagram to annotate the role played by a class or an operation in testing. The specified UML class diagrams, state machines, and activity diagrams constitute the executable test models. Based on them, we calculated the metrics (Section 7.1) to assess the cost-effectiveness and applicability of MoSH.

### 7.2.3 Testing SH-CPS with TM-Executor ($T_3$)

To answer RQ3, we applied TM-Executor to test a real SH-CPS (RAMA), based on the test model built in $T_2$. Section 7.3.2 gives a summary of the model. In total, ten functional behaviors and four self-healing behaviors were tested with five simulators of sensors and one simulator of an actuator. Via the simulators, ten environmental uncertainties (Table 12) were introduced during executions.

We used a single PC, with a processor of Intel Core i7 2.6 GHz and 16 GB of RAM, to run the experiment. During execution, timers were used to measure the time to execute the model, generate test data, evaluate constraints, and generate uncertainty values. For each guard, invariant, and uncertainty, we measured the *GenDataTime*, *EvalTime*, *GenUncTime* (Section 7.1) and conducted the Kruskal–Wallis test to check the effects of guard complexity, invariant complexity, and different ways of generating uncertainty values on the performance of TM-Executor.

## 7.3 Experiment Results

This section presents the results of each research question.

### 7.3.1 Results for RQ1

In total, we identified 832 instances of concepts from the specifications of nine SH-CPSs. After the two-steps refinement presented in Section 7.2.1, CMSU succeeded to cover all the identified concepts and their relationships. Table 8 summarizes the statistics of different concepts. As shown in the table, every system consists of one or more types of *PhysicalUnit*, which are constituted by *Sensor*s, *Actuator*s, and *Controller*s. All systems were designed for monitoring or controlling only one kind of *PhysicalProcess*, such as videoconferencing for VCS and traffic for TMS.

Table 8 Descriptive Statistics of the SH-CPSs

| Concept | VCS | TMS | APRS | RFID-SC | DSRL | ISR | RAMA | PeMS | VSS |
|---|---|---|---|---|---|---|---|---|---|
| **Self-Healing CPS** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **PhysicalProcess** | 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 |
| **Network** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **PhysicalUnit** | 7 | 5 | 5 | 4 | 8 | 2 | 2 | 4 | 2 |
| **Sensor** | 3 | 1 | 3 | 1 | 4 | 5 | 5 | 2 | 1 |
| **Controller** | 7 | 2 | 2 | 4 | 9 | 2 | 3 | 4 | 2 |
| **Actuator** | 2 | 0 | 5 | 0 | 4 | 1 | 1 | 0 | 0 |
| **Functional Behavior** | 4 | 1 | 1 | 1 | 4 | 1 | 10 | 7 | 3 |
| **Self-Healing Behavior** | 2 | 1 | 2 | 4 | 4 | 5 | 4 | 5 | 2 |
| **Goal** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **State** | 21 | 6 | 22 | 11 | 32 | 36 | 97 | 41 | 21 |
| **Probe** | 2 | 1 | 3 | 5 | 4 | 5 | 5 | 2 | 2 |
| **Effector** | 2 | 1 | 5 | 1 | 4 | 2 | 1 | 2 | 1 |
| **Measurement** | 2 | 1 | 5 | 5 | 7 | 4 | 4 | 5 | 2 |
| **Self-Diagnosis** | 2 | 1 | 5 | 4 | 4 | 5 | 4 | 5 | 2 |
| **Self-Recovery** | 4 | 3 | 3 | 4 | 4 | 5 | 9 | 5 | 3 |
| **Fault** | 2 | 1 | 3 | 4 | 4 | 5 | 4 | 5 | 2 |
| **Error** | 2 | 1 | 5 | 4 | 4 | 5 | 4 | 5 | 2 |
| **RecoveryPolicy** | 4 | 3 | 3 | 4 | 1 | 5 | 9 | 5 | 3 |
| **AdaptationAction** | 4 | 3 | 10 | 3 | 8 | 4 | 4 | 2 | 2 |
| **Uncertainty** | 3 | 1 | 4 | 4 | 5 | 6 | 10 | 6 | 1 |
| ***Total*** | 78 | 36 | 90 | 67 | 115 | 102 | 180 | 109 | 55 |

Table 9 presents the kinds of *Probes*, *RecoveryPolicies*, *Effectors* used by each system to diagnose or recovery from faults. *PhysicalProcessProbe* and *ControlEffector* are the most common *Probe* and *Effector* used by these SH-CPSs. With the respect of *RecoveryPolicy*, *ActionPolicy* is dominating. Totally, 39 *Uncertainties* were identified, 35 of which belong to Level 2 uncertainty. This reflects that the probability theory is the most common way to measure uncertainty.

Table 9 Descriptive Statistics of Categories of Probe, RecoveryPolicy, Effector, and Uncertainty

| Concept | | VCS | TMS | APRS | RFID-SC | DSRL | ISR | RAMA | PeMS | VSS | *P* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Probe** | **PerformanceProbe** | 2 | 0 | 0 | 3 | 0 | 0 | 0 | 2 | 2 | 31% |
| | **EventProbe** | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 10% |
| | **PhysicalProcessProbe** | 0 | 0 | 3 | 0 | 4 | 5 | 5 | 0 | 0 | 59% |
| **Recovery Policy** | **ActionPolicy** | 4 | 3 | 2 | 4 | 0 | 5 | 9 | 5 | 3 | 94% |
| | **GoalPolicy** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3% |
| | **UtilityFunctionPolicy** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 3% |
| **Effector** | **ParameterEffector** | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11% |
| | **ArchitectureEffector** | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 16% |
| | **ControlEffector** | 0 | 0 | 5 | 0 | 4 | 2 | 1 | 2 | 0 | 73% |
| **Uncertainty** | **Level 1** | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 5% |
| | **Level 2** | 3 | 1 | 3 | 2 | 5 | 6 | 8 | 6 | 1 | 87% |
| | **Level 3** | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 8% |
| | **Level 4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0% |
| | **Level 5** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0% |

*P* = n / N, where n is the number of occurrences of a subclass and N is the total number of occurrences of all sub-classes, e.g., *PerformanceProbe*, *EventProbe*, and *PhysicalProcessProbe* are subclasses of *Probe*.

### *7.3.2* Results for RQ2

To assess the additional effort required to apply MoSH to create executable test models, we report results of the *PerStereo* metric in Table 10. We applied stereotypes from MoSH to 15%, 15%, and 19% of model elements for RAMA, PeMS, and VSS. On average, 16% of model elements were stereotyped. This number gives us a rough indication of additional modeling effort required to use MoSH to create executable test models, as compared with applying standard UML notations.

To evaluate the applicability of MoSH, we first provide the statistics of model elements in Table 11. The statistics reflect the complexity of the test models for the three SH-CPSs. Among the three systems, RAMA is the most complicated one. It contains ten functional behaviors and four self-healing behaviors. In total, 377 model elements were used to specify the 14 behaviors. In contrast, PeMS and VSS are relatively simple. In total, 144 model elements were used for specifying 12 behaviors of PeMS and 97 model elements were used for specifying five behaviors of VSS.

Table 10 MoSH Evaluation Results (RQ2)

| Metric | RAMA | PeMS | VSS | *Avg.* |
|---|---|---|---|---|
| ***TotalElem*** | 377 | 144 | 97 | 206 |
| ***FunBeh*** | 10 | 7 | 3 | 7 |
| ***HealBeh*** | 4 | 5 | 2 | 4 |
| ***Diagnosis*** | 4 | 5 | 2 | 4 |
| ***Recovery*** | 9 | 5 | 3 | 6 |
| ***Uncertainty*** | 10 | 6 | 1 | 6 |
| ***PerStereo*** | 15% | 15% | 19% | 16% |

Second, we collected the numbers of functional behaviors, self-healing behaviors, fault diagnoses, fault recoveries, and uncertainties. As shown in Table 10 (the *FunBeh* and *HealBeh* rows) and Table 8 (the Functional Behavior and Self-Healing Behavior rows), all the identified functional and self-healing behaviors were captured in the test models. Moreover, self-diagnosis and self-recovery, the two key steps of self-healing behaviors, were also explicitly specified, as shown in the *Diagnosis* and *Recovery* rows in Table 10. They enable self-healing behaviors to be rigorously tested.

Table 11 Descriptive Statistics of Model Elements (RQ2)

| Element | RAMA | PeMS | VSS | *Avg.* |
|---|---|---|---|---|
| **Class** | 10 | 7 | 4 | 7 |
| **Attribute** | 42 | 12 | 11 | 21 |
| **Operation** | 29 | 14 | 11 | 18 |
| **Signal** | 10 | 10 | 4 | 8 |
| **Association** | 9 | 7 | 3 | 6 |
| **State Machine** | 14 | 12 | 5 | 10 |
| **State** | 97 | 41 | 21 | 53 |
| **Transition** | 166 | 41 | 38 | 81 |
| ***Total*** | 377 | 144 | 97 | 204 |

We specified ten uncertainties for RAMA, six uncertainties for PeMS, and one uncertainty for VSS, as shown in Table 12. With MoSH, we could define the universe,

categories, and measure for each uncertainty. Based on them, TM-Executor can introduce uncertainties via simulators/emulators, which enables the systems to be tested under uncertainties.

Table 12 Uncertainties in RAMA, PeMS and VSS (RQ2)

| SH-CPS | Uncertainty | Level | Universe | Measure | Categories |
|---|---|---|---|---|---|
| **RAMA** | **Wind Direction** | 3 | 0° ~ 360° | Possibility | Null |
| | **Wind Velocity** | 3 | 0 ~ 30 m/s | Possibility | Low, Med, High |
| | **GPS Noise** | 2 | -50 ~ 50 m | Probability | Null |
| | **Servo Deviation** | 2 | -1 ~ 1 $m/s^2$ | Probability | Null |
| | **Barometer Altitude Noise** | 1 | -10 ~ 10 m | N/A | N/A |
| | **Barometer Climb Rate Noise** | 2 | -0.5 ~ 0.5 $m/s^2$ | Probability | Null |
| | **Accelerometer Noise** | 2 | -1 ~ 1 $m/s^2$ | Probability | Null |
| | **Gyro Noise** | 2 | -0.1 ~ 0.1 rad/s | Probability | Null |
| | **Compass Noise** | 2 | -0.04 ~ 0.04 gauss | Probability | Null |
| | **Compass Hysteresis** | 2 | -0.01 ~ 0.01 gauss | Probability | Null |
| **PeMS** | **Vehicle Speed** | 3 | 10 ~ 120 km/h | Possibility | Null |
| | **Vehicle Size** | 3 | 2000 ~ 5000 L | Possibility | Mini, Compact, Mid, Large |
| | **Loop Detector Impedance** | 2 | 5 ~ 10 Ω | Probability | Null |
| | **Loop Detector Voltage** | 2 | 3 ~ 4 V | Probability | Null |
| | **Loop Detector Sensitivity** | 2 | 0.1 ~ 1 $\mu$H | Probability | Null |
| | **Loop Detector Latency** | 2 | 0 ~ 0.1 s | Probability | Null |
| **VSS** | **Latency of Channel** | 2 | 0 ~ 800 ms | Probability | Null |

### *7.3.3* Results for RQ3

To answer RQ3, we summarize the time spent by TM-Executor to perform testing activities in Table 13. On average, it took 5.6 seconds for traversing a transition, 39 milliseconds for test data generation, and less than one millisecond for exiting or entering a state, executing an operation, evaluating a constraint or generating an uncertainty value. During test execution, most of the time was spent on traversing transitions. As shown in Table 13, after sending a stimulus to the SH-CPS, the system maximally took 9.5 seconds to perform an instructed operation and enter a target state. However, this time is determined by the software and simulators/emulators of the system. Due to the high computational cost of executing the software and the simulators/emulators, changing the state of the software takes considerable time. To improve the efficiency of testing, a

Table 13 Evaluation Result (RQ3)

| Metric | Mean (ms) | Minimum (ms) | Maximum (ms) |
|---|---|---|---|
| ***TraTime*** | 5607 | 5260 | 9531 |
| ***ExitStateTime*** | <1 | <1 | <1 |
| ***EnterStateTime*** | <1 | <1 | <1 |
| ***ExeOpTime*** | <1 | <1 | 1 |
| ***GenDataTim*e** | 39 | 3 | 270 |
| ***EvalTime*** | <1 | <1 | 1 |
| ***GenUncTime*** | <1 | <1 | <1 |

distributed testing framework can be developed in the future. It may reduce the time cost of executions, by executing testing components in separate computational nodes.

For the first Kruskal-Wallis test which aims to evaluate the effect of guard complexity on the *GenDataTime*, the p-value is less than $2.2 \times 10^{-16}$. Thus, we conclude that the *GenDataTime*s were significantly different for guards with different complexities. On average, TM-Executor took 7.2, 9.3, 17.1, and 63.3 milliseconds to generate test data for the four groups of guards. For the second Kruskal-Wallis test which aims to evaluate the effect of invariant complexity on the *EvalTime*, its p-value is 0.6, implying that the complexity of the invariants did not significantly affect the time taken for evaluation. For the third Kruskal-Wallis test which aims to evaluate the effect of different ways of generating uncertainty values on *GenUncTime*, its p-value equals to 0.6. It reflects that the different ways of generating uncertainty values did not significantly influence the time spent on uncertainty generation.

In the experiment, one fault (state invariant violation) was detected by TM-Executor. This fault led to the collision of the drone. Though a self-healing behavior helped the drone automatically avoid collisions with other vehicles, the drone failed to keep a safe distance from an intruding vehicle because of uncertainties from the GPS, compass, barometer, accelerometer, gyro, and servos. Note that we do not assume the system was correctly implemented and we only aim to test whether the self-healing behaviors can successfully recover systems from faults under uncertainties.

## 7.4 Overall Discussion

In this section, we provide an overall discussion based on the results presented in Section 7.3. Based on the results presented in Section 7.3.1, we conclude that our conceptual model (CMSU) covered all concepts and their relations identified in nine SH-CPSs. This increases our confidence that CMSU captures the main concepts of SH-CPSs and uncertainties.

Based on the results presented in Section 7.3.2, we conclude that, on average, it needed additional 16% of modeling effort to apply MoSH to create the executable test models, as compared with applying standard UML notations, for the selected SH-CPSs. We demonstrated the applicability of MoSH to specify executable test models for three diverse SH-CPSs of varying complexities. This effort gives us evidence that MoSH is capable of

modeling different SH-CPSs to support uncertainty-aware executable model-based testing. For the current evaluation, the first author of the paper created all the models. We acknowledge that a better evaluation would be to conduct a controlled experiment with more modelers to assess the applicability of MoSH. Conducting such controlled experiments (even in an academic setting) requires resources and opportunities, whereas we are actively pursuing such opportunities.

Test models developed with MoSH can be executed by TM-Executor to enable the executable model-based testing. By applying TM-Executor to test a real-world SH-CPS, we demonstrated that TM-Executor could automatically test the SH-CPS under uncertainties. Moreover, TM-Executor could obtain runtime information about the system's actual behaviors from test execution. The information can be used by dedicated testing strategies [12, 49] to find the optimal sequence of testing stimuli for fault detection. However, devising such strategies is not covered in this paper. In the evaluation reported in this paper, we applied a random strategy to guide the selection of stimuli, without exploiting the runtime information provided by TM-Executor. Thus, the random strategy is not optimal regarding fault detection. We have devised a more effective strategy and applied it with TM-Executor to test three SH-CPSs reported in [12]. With the help of runtime information provided by TM-Executor, the new strategy detected significantly more faults than a traditional coverage based method [12]. Based on the results presented in Section 7.3.3, we also conclude that the time taken by TM-Executor to perform testing activities was relatively small, i.e., in the order of milliseconds. Thus, it is practicable to apply TM-Executor to perform the executable model-based testing.

## 7.5 Threats to Validity

*Conclusion validity threats* are related to factors that can affect conclusions drawn from experimental results. The random testing strategy implemented in TM-Executor leads to different behaviors of a system being tested. For different behaviors, the amount of time spent by TM-Executor to perform testing activities varies as well. To deal with such a threat, we ran the experiment 100 runs.

*External validity threats* concern the generalization of the results. One *external validity threat* is that we only applied nine SH-CPSs to evaluate CMSU and applied three of them to evaluate MoSH. Although the nine systems are from different domains, they cannot

assure to cover all kinds of self-healing behaviors. Based on the specifications of the nine systems, 40 uncertainties were identified. Among them, 17 and 10 uncertainties were used to evaluate MoSH and TM-Executor respectively. Although the uncertainties cover the three levels (Level 1 to Level 3) of uncertainties that are supported by MoSH and TM-Executor, more uncertainties at each level are still needed to generalize the results further. Another *external validity threat* is that only one system was employed to evaluate TM-Executor's performance. However, diverse uncertainties, state invariants, guards, operations were exploited by TM-Executor to test a real SH-CPS. Nonetheless, additional case studies are needed to generalize the results further.

*Construct validity threats* refer to the degree to which the experiment setting (including two metrics for the CMSU evaluation, seven metrics for MoSH and eight metrics for TM-Executor evaluation) reflects the construct under study (i.e., the quality of CMSU, the cost-effectiveness of MoSH and the performance of TM-Executor). To reduce the threats, we carefully selected and defined the metrics focusing on our overall objective of testing SH-CPSs under uncertainties. Another construct validity threat is that the levels of uncertainties used in the evaluation were manually classified. To avoid incorrect classification, we did the classification based on detailed product specifications. Nevertheless, additional metrics and other ways of evaluation are also possible.

# 8 Related work

In this section, we discuss related work concerning fault-tolerant computing (Section 8.1), the conceptual model of SH-CPSs and uncertainties (Section 8.2), modeling methods for SH-CPSs (Section 8.3), and testing approaches for SH-CPSs under uncertainties (Section 8.4). Section 8.5 summarizes how our work advances state of the art.

## 8.1 Fault-Tolerant Computing

A self-healing system can perceive that it is not operating correctly and, without human intervention, make necessary changes to its architecture or behaviors to restore itself to a normal state [2]. Since the goal of self-healing is to make system fault-tolerant, self-healing can be seen as a kind of fault-tolerant mechanism. However, not all fault-tolerant mechanisms can be considered as self-healing behaviors, as many fault-tolerant

mechanisms can only mask faults and they cannot dynamically change a system's architecture or behaviors to recover the system from faults [3].

Fault-tolerance is defined as "the ability of a system to continuously perform its intended functions in the presence of a given number of faults" [50]. A certain amount of redundancy has to be applied to achieve fault-tolerance, including time redundancy, information redundancy, hardware redundancy, and software redundancy [51]. In time redundancy, computation or data transition is performed multiple times to overcome transient faults. Information redundancy uses error-detection code or error-correction code to detect or mask faulty information caused by incorrect storage or transition. For hardware redundancy, computation is performed on several instances of hardware components simultaneously. By comparing their outputs and voting on the result, the fault can be detected and masked. Similarly, multi-version software fault-tolerance techniques employ redundant software modules to mask faults. As the aforementioned fault-tolerance techniques cannot make runtime adaptation that aims to restore normal system operations, they are not regarded as self-healing behaviors. In contrast, single-version software fault-tolerance techniques enable the software to detect faults, diagnose causes and prevent the propagation of their effect throughout the system. Thus, the single version techniques can be considered as a kind of self-healing behaviors, whereas traditional single version techniques mainly rely on checkpoints and roll-backward or roll-forward to handle a detected fault [52]. In contrast, the self-healing behaviors, targeted in this paper, can also modify a system's architecture or behaviors in case of faults. Thus, the term "self-healing" is used in this paper to represent the new kind of fault-tolerant mechanisms.

## 8.2 Concepts of CPS, Self-Healing, and Uncertainty

After a decade's effort, key elements of CPSs and self-healing systems have been identified by academic and industrial communities. Zhang et al. [53] defined a CPS as a set of heterogeneous physical units communicating via heterogeneous networks, where physical units are recognized as the first-class objects of CPSs and networks are also considered important, as they enable communication among physical units. Lee et al. [54] considered sensors and actuator as interfaces between computational and physical components. CPSs were characterized by integrating computation and physical processes [55], and their primary goal is to control physical processes efficiently [56]. For self-

healing, fault diagnosis and recovery are identified as its key steps by Psaier et al. [57]. Kephart et al. [23] and White et al. [58] introduced and evaluated three types of recovery policies, particularly for self-healing behaviors. Moreover, inspired by goal-oriented self-healing approaches [59], the goal of self-healing behaviors is also captured. We adopted these key concepts in the components of CPSs (Section 4.1) and self-healing behavior (Section 4.2) parts of our conceptual model.

How to cope with uncertainty is a grand challenge [1]. In the past, the effort was mostly spent on identifying uncertainty sources in SH-CPSs. Ramirez et al. [60] proposed a taxonomy of uncertainty sources in dynamically adaptive systems at the requirement, design, and execution phases. The uncertainty taxonomy is given from a system designer's perspective, and it aims to help the designer to mitigate the effect of uncertainties. On the contrary, we define the uncertainty from a tester's perspective, and we concern more about how to specify uncertainties and simulate their effect based on the specified uncertainties. Esfahani et al. [4] proposed another nine uncertainty sources, which need to be considered during design for self-adaptive systems. Although this paper introduces probability and possibility theories to model uncertainties, it does not identify the key elements, i.e., probability, possibility, and necessity, that are required to be specified for modeling uncertainties. Zhang et al. [53] developed a conceptual model of uncertainty for CPSs. The conceptual model captures three kinds of measure, i.e., vagueness, probability, and ambiguity. However, the key elements of the three kinds of measure are not provided, such as the universe and membership function. Alternatively, in the uncertainty part of the conceptual model (Section 4.3), we captured the concepts of *Universe*, *Category*, *MembershipFunction*, *Probability, Possibility*, and *Necessity* to specify uncertainty.

## 8.3 UML-based Modeling for SH-CPSs

To tackle the complexity of CPSs, researchers proposed to adopt model-based engineering [61], which uses models to facilitate system design, development, verification, and validation. Since a CPS is an integration of computation and physical processes, it is typically modeled as a hybrid system, where physical processes are specified as continuous models, and computation parts are defined as discrete models [62]. For physical processes, several modeling tools can be used to specify the continuous models such as Simulink [63], Modelica [64], and SystemC [65]. For the computational part, UML is the most broadly

used modeling language, and several UML profiles have been developed to extend UML for modeling CPSs, fault-tolerant mechanisms, or testing components.

**Sys**tems **M**odeling **L**anguage (SysML) [6] reduces UML's software-centric restrictions and adds new notations and diagram types to model a broad range of components, including hardware, software, data, and physical entities. However, SysML still lacks support from standards to be executable, and model transformation is needed to make SysML models executable [66]. Since our work aims to realize an executable model-based testing approach, the test model has to be executable. Thus, we chose to use fUML as the starting point to build our solution. xUML [67] is another choice for creating executable models. However, the semantics of modeling notations used by xUML are quite different than UML. As fUML is more broadly used [68], we chose to use fUML in our work.

The profile of MARTE [7] adds capabilities to UML for model-driven development of real-time embedded systems. Particularly, it concerns how to model analyses and designs of real-time and embedded systems. We, however, aim to test if an SH-CPS can successfully detect and recover from faults under uncertainties. To fulfill the aim, we provide additional modeling support for self-healing behaviors and uncertainties. MARTE provides the capability of specifying probability distributions regarding frequencies, but not for modeling uncertainties due to incomplete or imprecise knowledge (Section 2.2). For instance, the measurement noise of a GPS could be described as Low, Medium, and High, but the boundaries of the three types are not precisely defined. In this case, testers can apply membership functions, provided by our modeling framework (Section 5.3.1) to specify the uncertainties associated with the measurement noise. Our modeling framework also reused the probability distribution library from MARTE, as the library provides well-defined datatypes for modeling distributions.

The profile of **D**ependability **A**nalysis **M**odeling (DAM) [8] extends MARTE for dependability modeling and analysis. Regarding self-healing behaviors, the profile provides «DaReplacementStep» and «DaReallocationStep» to model replacement and reallocation performed by software components to recover from faults. Besides replacement and reallocation, SH-CPSs can also use runtime reconfiguration and adaptations to handle detected faults, directed by different kinds of recovery policies. To support modeling these fault recovery mechanisms (identified in Section 4.2.2), we provide the SH-CPS Behaviors Profile and an accompanying modeling methodology in Section 5.2.

Modeling **Q**uality Of Service And **F**ault-**T**olerance Characteristics And Mechanisms **P**rofile (QFTP) [9] extends UML to model fault-tolerant software architecture. This profile focuses on modeling the redundancy used by the fault-tolerant mechanisms, including policies to create, deploy, monitor, and activate replicas, whereas it cannot be used to model self-healing behaviors that do not rely on replicas to detect and recover from faults. Alternatively, we identified three kinds of approaches for fault detection (Section 4.2.1) and three kinds of fault recovery policies (Section 4.2.2) that are not specifically associated with replicas. In Section 5.2, we provide the SH-CPS Behaviors Profile and a guideline to apply the profile to model the logic of fault detection and recovery.

UTP v2 [29] provides dedicated modeling support for model-based testing. It covers a variety of concepts that are deemed mandatory for testing, such as test-specific actions, test data, and test verdicts. Our modeling framework reuses the stereotypes from the Test Architecture and Test Behavior packages of UTP to specify testing components, as these stereotypes are standardized and have been precisely defined by UTP.

## 8.4 Testing SH-CPSs Under Uncertainties

This section discusses existing approaches for testing self-healing behaviors (Section 8.4.1) and for testing systems under uncertainties (Section 8.4.2).

### 8.4.1 Testing Self-Healing Behaviors

Fault injection is a straightforward method to test the recovery mechanisms of self-healing systems. By introducing faults, self-healing behaviors can be exercised; otherwise, they are rarely triggered. A fault model can be used to capture the faults that are to be handled by self-healing systems. Gama et al. [42] proposed a fault model that captures five types of faults (i.e., application hang, component crash, stale service, denial of service and excessive thread allocation). According to the model, faults are introduced by deploying and activating faulty Java Beans to trigger self-healing behaviors. Huebscher et al. [69] proposed to use context models to simulate sensor data and use the simulated data to trigger and evaluate self-healing behaviors. Similarly, Hänsel et al. [70] used a simulated architecture model as input to test the feedback loop of a self-healing system.

These testing approaches focus on the methods to trigger self-healing behaviors, whereas they do not provide solutions to determine when to trigger the behaviors. Alternatively, we propose to build an executable test model capturing both functional and self-healing

behaviors, together with uncertainties. By executing the model and the system under test together, TM-Executor can dynamically decide how to exercise the system's self-healing behaviors under uncertainties.

### 8.4.2 Testing Systems Under Uncertainties

Uncertainty in CPSs is still immature [5], whereas researchers have realized the importance to verify and validate SH-CPSs under uncertainties [1, 4, 60, 71]. A model checking approach has been proposed by Yang et al. [72] to formally verify the correctness of self-adaptation in the presence of uncertainties. However, formal verification approaches suffer from the scalability issue for realistic, complex systems [54, 73]. Besides formal verification, another option is testing. Fredericks et al. [74] proposed a runtime feedback loop (MAPE-T) to test adaptive systems under runtime uncertainty. Based on the feedback loop, they developed a runtime testing framework [75] that dynamically adapts test cases to ensure that a system continues to behave correctly in uncertain environmental conditions. However, this work does not concern how to generate initial test cases and how to exercise system behaviors. Alternatively, our testing approach uses an executable test model to capture expected system behaviors. By executing the model together with the system under test, we dynamically test the system against the model.

As it is expensive to test SH-CPSs in a real environment, simulation becomes necessary. Ramirez et al. [76] proposed to use noisy values from sensors in simulations to find the combinations of noises that can reveal faults, whereas this approach supports only three types of uncertainty (i.e., static, periodic, and sporadic noises). Similarly, Minnerup et al. [77] presented an error model to capture inaccuracies of actuators that are specific for autonomous vehicles. Based on the error model, samples of inaccurate actuation are used in simulations to test the vehicles under uncertainties. Compared with these two works, we present a more general approach to support a broader range of uncertainties. Particularly, we provide SH-CPS Uncertainty Profile (Section 5.3) to specify uncertainties. Based on the specified uncertainties, TM-Executor can automatically introduce uncertainties into interactions between SH-CPSs and their environments to enable uncertainty-aware testing.

Model-based testing is another enabler for uncertainty-aware testing. Zhang et al. [5, 78, 79] recently proposed a complete model-based approach for uncertainty-aware testing of CPSs. In this approach, a system's uncertain, expected behaviors are specified as UML state machines, based on which test cases are generated and executed. However, the

approach is not suitable to be applied to test SH-CPSs. First, for SH-CPSs, generating test cases offline is challenging, as it is difficult to predict runtime adaptations performed by self-healing behaviors. Second, it is hard to determine coverage criteria and choose a proper set of paths that have to be covered by test cases. Alternatively, MoSH provides the modeling constructs and methodology to specify executable test models. With TM-Executor, a specified test model can be executed together with the system under test. Meanwhile, uncertainties specified in the model are automatically introduced into the testing environment, allowing the system to be dynamically tested against the model in the presence of uncertainties.

## 8.5 Summary

Our work advances state of the art in several ways. First, though some works define self-healing concepts [2, 57] and uncertainty related concepts [27, 60], there is no single work that jointly captures concepts of self-healing and uncertainty in the context of CPSs. To this end, we present the conceptual model of SH-CPSs and uncertainties. It is built on the literature [23, 53-56, 58, 80-82] to conceptualize self-healing behaviors of CPSs and uncertainty together. Second, though there exist several modeling notations for defining self-healing behaviors such as [8, 9]; however, none of them provide a complete modeling solution to create the executable test model for testing SH-CPSs under uncertainties. Based on existing standards, MoSH provides an integrated modeling solution to provide such supports. Third, although there exist adaptive test strategies [75, 83] to test self-healing behaviors, there is no evidence that such strategies can be adapted to test SH-CPSs in the presence of environmental uncertainties. TM-Executor provides an integrated testing environment for testing the SH-CPSs under uncertainties. Finally, we extend existing model-based testing [5, 78, 79] to executable model-based testing that is underpinned by MoSH and TM-Executor. The new approach can provide information about a system's actual behaviors, obtained from execution. The information can be used to effectively detects faults, which has been demonstrated by our recently published work [12].

# 9 Conclusion and Future Work

**S**elf-**H**ealing **C**yber-**P**hysical **S**ystems (SH-CPSs) have the built-in capability to detect and recover from faults by themselves at runtime. Since such systems are often operated in

highly unpredictable environments and affected by various uncertainties, these systems are required to deal with such uncertainties even during the process of fault diagnosis and recovery. To effectively test the SH-CPSs under uncertainties, we proposed an executable model-based testing approach. As the first step towards realizing the approach, we present a **M**odeling Framework **o**f **SH**-CPSs (MoSH) and a **T**est **M**odel **Executor** (TM-Executor) in this paper. They provide the modeling support and execution environment for the new testing approach. MoSH and TM-Executor were evaluated with several SH-CPSs. Based on the evaluation, we can conclude that 1) executable test models could be successfully created with MoSH to capture expected behaviors and uncertainties for the selected systems and 2) TM-Executor could dynamically test the systems against the models, by executing them together. Information about the system's actual behaviors can be obtained from the execution. However, to fully leverage the information, dedicated testing strategies have to be devised. On the one hand, the testing strategies should use the information to learn the optimal sequence of test actions that have the highest chance to reveal a fault. On the other hand, the information should be used to effectively find a sequence of uncertainty values that can work together with the test actions to make the system fail. Regarding the first aspect, we have devised a fragility-oriented strategy [12] to find the optimal invocations. However, the uncertainty values currently are only randomly generated. Thus, a more advanced uncertainty generation strategy is required to detect faults under uncertainties. When the strategy is devised, the proposed executable model-based testing approach can be fully realized, and we will perform an extensive empirical study to assess the fault detection ability of the whole approach.

# References


[1]. Bures, T., Weyns, D., Berger, C., Biffl, S., Daun, M., Gabor, T., Garlan, D., Gerostathopoulos, I., Julien, C., Krikava, F.: 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, vol.40, pp.28-32 (2015)

[2]. Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S.: Self-healing systems—survey and synthesis. Decision Support Systems, vol.42, pp.2164-2185 (2007)

[3]. Rodosek, G.D., Geihs, K., Schmeck, H., Stiller, B.: Self-Healing Systems: Foundations and Challenges. Self-Healing and Self-Adaptive Systems, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)

[4]. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. Software Engineering for Self-Adaptive Systems II, pp. 214-238. Springer (2013)

[5]. Zhang, M., Ali, S., Yue, T., Norgren, R., Okariz, O.: Uncertainty-Wise Cyber-Physical System test modeling. Software & Systems Modeling, https://doi.org/10.1007/s10270-017-0609-6 (2017)

[6]. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language. Morgan Kaufmann (2014)

[7]. OMG: Profile for modeling and analysis of real-time and embedded systems (MARTE). formal/2011-06-02 (2011)

[8]. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Software & Systems Modeling, vol.10, pp.313-336 (2011)

[9]. OMG: Profile for modeling quality of service and fault tolerance characteristics and mechanisms. formal/2008-04-05 (2008)

[10]. OMG: Semantics Of A Foundational Subset For Executable UML Models V1.2.1. formal/2016-01-05 (2016)

[11]. OMG: Precise Semantics Of UML State Machines (PSSM). 1.0 - Beta 1 (2017)

[12]. Ma, T., Ali, S., Yue, T., Elaasar, M.: Fragility-oriented testing with model execution and reinforcement learning. In: IFIP International Conference on Testing Software and Systems, pp. 3-20 (2017)

[13]. Holub, O., Hanzálek, Z.: Low-cost reconfigurable control system for small UAVs. IEEE Transactions on Industrial Electronics, vol.58, pp.880-889 (2011)

[14]. Selic, B.: A systematic approach to domain-specific language design using UML. In: Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on, pp. 2-9 (2007)

[15]. Giachetti, G., Marín, B., Pastor, O.: Integration of domain-specific modelling languages and UML through UML profile extension mechanism. IJCSA, vol.6, pp.145-174 (2009)

[16]. do Nascimento, L.M., Viana, D.L., Neto, P.A.S., Martins, D.A., Garcia, V.C., Meira, S.R.: A systematic mapping study on domain-specific languages. In: Proceedings of

the 7th International Conference on Software Engineering Advances (ICSEA'12), pp. 179-187 (2012)

[17]. Robert, S., Gérard, S., Terrier, F., Lagarde, F.: A lightweight approach for domain-specific modeling languages design. In: Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on, pp. 155-161 (2009)

[18]. Ma, T., Ali, S., Yue, T.: Modeling Healing Behaviors of Cyber-Physical Systems with Uncertainty to Support Automated Testing. Simula Research Lab (2016)

[19]. Blanke, M., Schröder, J.: Diagnosis and fault-tolerant control. Springer (2006)

[20]. Venkatasubramanian, V., Rengaswamy, R., Yin, K., Kavuri, S.N.: A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. Computers & chemical engineering, vol.27, pp.293-311 (2003)

[21]. Siripongwutikorn, P., Banerjee, S., Tipper, D.: A survey of adaptive bandwidth control algorithms. Communications Surveys & Tutorials, IEEE, vol.5, pp.14-26 (2003)

[22]. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: Proceedings of the first workshop on Self-healing systems, pp. 27-32 (2002)

[23]. Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on, pp. 3-12 (2004)

[24]. Koutsoumpas, V.: A model-based approach for the specification of a virtual power plant operating in open context. In: Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems, pp. 26-32 (2015)

[25]. Simmonds, J., Ben-David, S., Chechik, M.: Monitoring and recovery of web service applications. The smart internet, pp. 250-288. Springer (2010)

[26]. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, pp. 2-8 (2006)

[27]. Walker, W.E., Lempert, R.J., Kwakkel, J.H.: Deep uncertainty. Encyclopedia of operations research and management science, pp. 395-402. Springer (2013)

[28]. Dubois, D., Prade, H.: Possibility theory: an approach to computerized processing of uncertainty. Springer Science & Business Media (2012)

[29]. OMG: UML Testing Profile. ptc/17-09-29 (2017)

[30]. OMG: Unified Modeling Language V2.5. formal/15-03-01 (2015)

[31]. (OMG), O.M.G.: Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF). (2013)

[32]. Sivanandam, S., Sumathi, S., Deepa, S.: Introduction to fuzzy logic using MATLAB. Springer (2007)

[33]. Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D.: Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference, pp. 173-184 (2012)

[34]. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), pp. 1-4 (2009)

[35]. Tatibouet, J.: Moka – A simulation platform for Papyrus based on OMG specifications for executable UML. In: EclipseCon, (2016)

[36]. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Transactions on software engineering, vol.39, pp.1376-1402 (2013)

[37]. Dubois, D., Prade, H., Sandri, S.: On possibility/probability transformations. Fuzzy logic, pp. 103-112. Springer (1993)

[38]. Broman, D., Brooks, C., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of FMUs for co-simulation. In: Proceedings of the Eleventh ACM International Conference on Embedded Software, pp. 2 (2013)

[39]. Cellier, F.E., Kofman, E.: Continuous system simulation. Springer Science & Business Media (2006)

[40]. Ali, S., Briand, L.C., Hemmati, H.: Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. Software & Systems Modeling, vol.11, pp.633-670 (2012)

[41]. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 202-207 (2011)

[42]. Gama, K., Donsez, D.: Deployment and activation of faulty components at runtime for testing self-recovery mechanisms. ACM SIGAPP Applied Computing Review, vol.14, pp.44-54 (2014)

[43]. Cioara, T., Anghel, I., Salomie, I., Dinsoreanu, M., Copil, G., Moldovan, D.: A reinforcement learning based self-healing algorithm for managing context adaptation. In: Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services, pp. 859-862 (2010)

[44]. Park, J., Lee, S., Yoon, T., Kim, J.M.: An autonomic control system for high-reliable CPS. Cluster Computing, vol.18, pp.587-598 (2015)

[45]. Staszesky, D., Craig, D., Befus, C.: Advanced feeder automation is here. IEEE Power and Energy Magazine, vol.3, pp.56-63 (2005)

[46]. Lu, X.-Y., Varaiya, P., Horowitz, R., Palen, J.: Faulty loop data analysis/correction and loop fault detection. In: 15th World Congress on Intelligent Transport Systems and ITS America's 2008 Annual Meeting, (2008)

[47]. Ryu, B.-H., Jeon, D., Kim, D.-H.: A Robust Video Streaming Based on Primary-Shadow Fault-Tolerance Mechanism. In: International Conference on Ubiquitous Computing and Multimedia Applications, pp. 66-75 (2011)

[48]. Yue, T., Ali, S.: Empirically evaluating OCL and Java for specifying constraints on UML models. Software & Systems Modeling, vol.15, pp.757-781 (2016)

[49]. Veanes, M., Roy, P., Campbell, C.: Online testing with reinforcement learning. Formal Approaches to Software Testing and Runtime Verification, pp.240-253 (2006)

[50]. Nelson, V.P.: Fault-tolerant computing: Fundamental concepts. Computer, vol.23, pp.19-25 (1990)

[51]. Dunrova, E.: Fault Tolerant Design: An Introduction. Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden, (2008)

[52]. Torres-Pomales, W.: Software fault tolerance: A tutorial. NASA Langley Research Center; Hampton, VA United States (2000)

[53]. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model. In: Modelling Foundations and Applications: 12th European Conference, ECMFA (2015)

[54]. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. Lee & Seshia (2011)

[55]. Shi, J., Wan, J., Yan, H., Suo, H.: A survey of cyber-physical systems. In: Wireless Communications and Signal Processing (WCSP), 2011 International Conference on, pp. 1-6 (2011)

[56]. Sridhar, S., Hahn, A., Govindarasu, M.: Cyber–physical system security for the electric power grid. Proceedings of the IEEE, vol.100, pp.210-224 (2012)

[57]. Psaier, H., Dustdar, S.: A survey on self-healing systems: approaches and systems. Computing, vol.91, pp.43-73 (2011)

[58]. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: null, pp. 2-9 (2004)

[59]. Morandini, M., Penserini, L., Perini, A.: Automated mapping from goal models to self-adaptive systems. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 485-486 (2008)

[60]. Ramirez, A.J., Jensen, A.C., Cheng, B.H.: A taxonomy of uncertainty for dynamically adaptive systems. In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) pp. 99-108 (2012)

[61]. Ramos, A.L., Ferreira, J.V., Barceló, J.: Model-based systems engineering: An emerging approach for modern systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol.42, pp.101-111 (2012)

[62]. Derler, P., Lee, E., Vincentelli, A.S.: Modeling cyber–physical systems. Proceedings of the IEEE, vol.100, pp.13-28 (2012)

[63]. Dabney, J.B., Harman, T.L.: Mastering simulink. Pearson/Prentice Hall (2004)

[64]. Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., Broman, D., Sandholm, A.: OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In: IEEE International Symposium on Computer-Aided Control Systems Design, pp. 1588-1595 (2006)

[65]. Black, D.C., Donovan, J., Bunton, B., Keist, A.: SystemC: From the ground up. Springer Science & Business Media (2011)

[66]. Fritzson, P., Rouquette, N.F., Schamai, W.: An Overview of the SysML-Modelica Transformation Specification. (2010)

[67]. Carter, K.: Executable UML (xUML). (2007)

[68]. Mayerhofer, T.: Testing and debugging UML models based on fUML. In: Software Engineering (ICSE), 2012 34th International Conference on, pp. 1579-1582 (2012)

[69]. Huebscher, M.C., McCann, J.A.: Simulation model for self-adaptive applications in pervasive computing. In: Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on, pp. 694-698 (2004)

[70]. Hänsel, J., Vogel, T., Giese, H.: A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models. In: Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2015 IEEE International Conference on, pp. 134-139 (2015)

[71]. Ali, S., Lu, H., Wang, S., Yue, T., Zhang, M.: Uncertainty-Wise Testing of Cyber-Physical Systems. Advances in Computers, vol. 107, pp. 23-94. Elsevier (2017)

[72]. Yang, W., Xu, C., Liu, Y., Cao, C., Ma, X., Lu, J.: Verifying self-adaptive applications suffering uncertainty. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 199-210 (2014)

[73]. Zheng, X., Julien, C., Kim, M., Khurshid, S.: On the state of the art in verification and validation in cyber physical systems. The University of Texas at Austin, The Center for Advanced Research in Software Engineering, Tech. Rep. TR-ARiSE-2014-001, vol.1485, (2014)

[74]. Fredericks, E.M., Ramirez, A.J., Cheng, B.H.: Towards run-time testing of dynamic adaptive systems. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 169-174 (2013)

[75]. Fredericks, E.M., Cheng, B.H.: Automated generation of adaptive test plans for self-adaptive systems. In: Appear in Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, pp. 157-168 (2015)

[76]. Ramirez, A.J., Jensen, A.C., Cheng, B.H., Knoester, D.B.: Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pp. 568-571 (2011)

[77]. Minnerup, P., Knoll, A.: Testing Automated Vehicles against Actuator Inaccuracies in a Large State Space. IFAC-PapersOnLine, vol.49, pp.38-43 (2016)

[78]. Zhang, M., Li, Y., Ali, S., Yue, T.: Uncertainty-Wise and Time-Aware Test Case Prioritization with Multi-Objective Search. Technical Report, 2017-03, Simula Research Lab, Norway (2017) https://www.simula.no/publications/uncertainty-wise-and-time-aware-test-case-prioritization-multi-objective-search

[79]. Zhang, M., Ali, S., Yue, T.: Uncertainty-wise Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach. Technical Report 2016-13, Simula Research Lab, Norway (2017) https://www.simula.no/publications/uncertainty-based-test-case-generation-and-minimization-cyber-physical-systems-multi

[80]. NSF: Cyber Physical Systems. NSF 14-542 (2014)

[81]. Kim, K.-D., Kumar, P.R.: Cyber–physical systems: A perspective at the centennial. Proceedings of the IEEE, vol.100, pp.1287-1308 (2012)

[82]. Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. Dependable and Secure Computing, IEEE Transactions on, vol.1, pp.11-33 (2004)

[83]. Lahami, M., Krichen, M., Jmaiel, M.: Safe and efficient runtime testing framework applied in dynamic and distributed systems. Science of Computer Programming, vol.122, pp.1-28 (2016)

# Appendix A. Execution Process of an Executable Test Model

This appendix presents an activity diagram to illustrate the execution process of an executable test model.

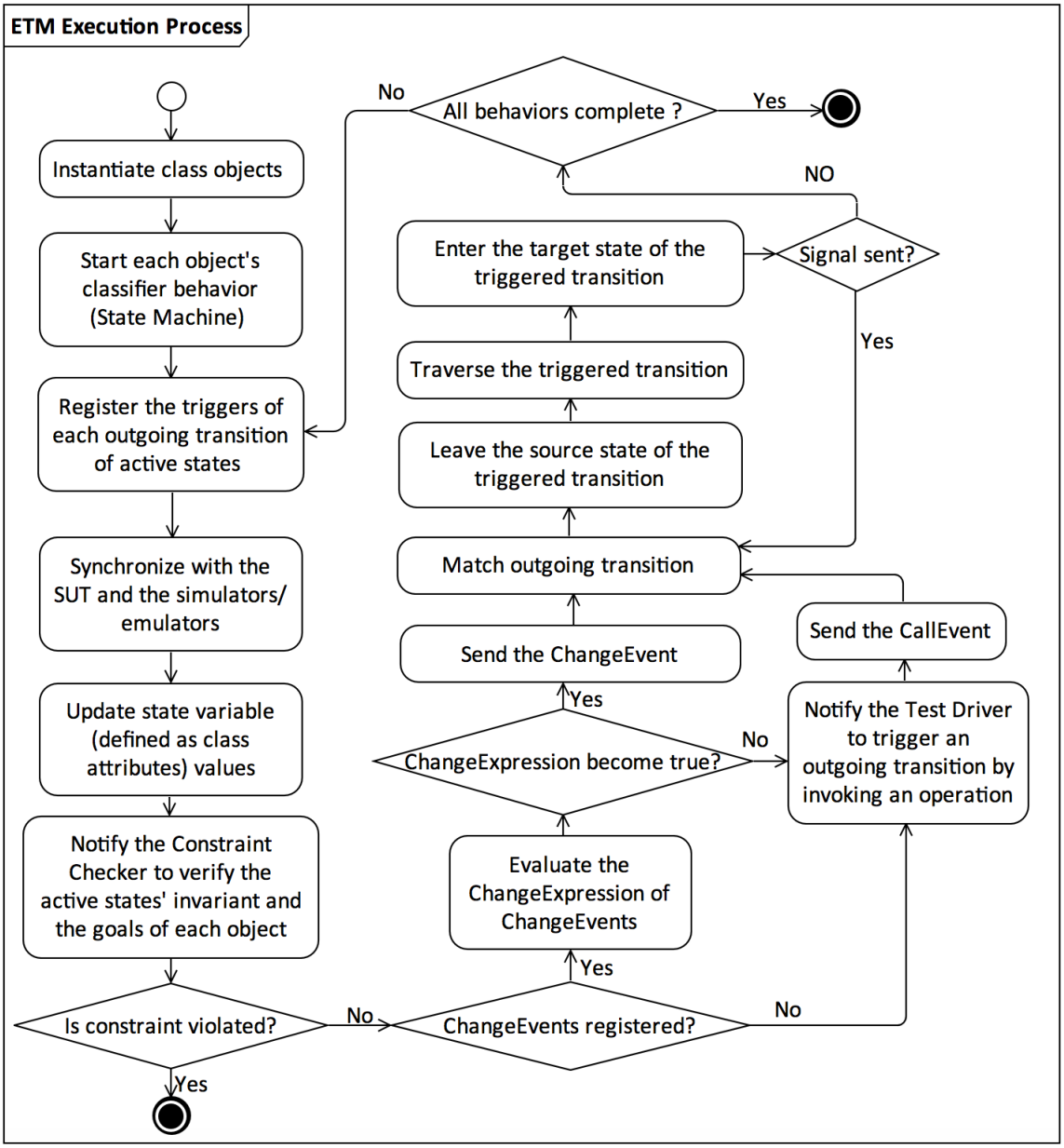


Figure 23 Execution Process of an Executable Test Model

# Appendix B. Extensions to fUML and PSSM

To facilitate executable model-based testing, we made several extensions to fUML and PSSM, and they are summarized below.

Table 14 Summary of the Extensions to fUML and PSSM

| Extension | UML Metaclass | Execution Model Element | Execution Semantics |
|---|---|---|---|
| Extensions for stereotypes | Operation stereotyped with «CreateStimulus Action» | CreateStimulusAction Execution | Invoke the testing interface corresponding to the URI defined in the opaque behavior of the operation, taking input parameter values as inputs. |
| | Operation stereotyped with «CheckProperty Action» | CheckPropertyAction Execution | Invoke the testing interface corresponding to the URI defined in the opaque behavior of the operation. Update attributes values using the outputs of the testing interface. |
| Extensions for additional metaclasses | BroadcastSignal Action | BroadcastSignalAction Activation | Construct a signal using the values from argument pins and send the signal to all objects that are associated with the object from the source pin. |
| | ChangeEvent | ChangeEventOccurrence | The change expression of the change event is evaluated, whenever related attributes' values are updated. If the change expression becomes true, the change event occurs. |
| Extensions to existing semantics | State | StateActivation | Besides the semantics defined in PSSM, when the state is entered, the invariant of the state is registered to the Constraint Checker. |
| | ExternalTransition | ExternalTransitionActivation | Besides the semantics defined in PSSM, the target state of the transition can only be entered when the target state's invariant becomes true. |
| | LocalTransition | LocalTransitionActivation | |
| | InternalTransition | InternalTransitionActivation | |
| Extensions to OpaqueAction | OpaqueAction | InputTestAPIInvocation Activation | Invoke the testing interface, whose URI is specified in the opaque expression of the OpaqueAction, taking the values from its input pins as input. |
| | | OutputTestAPIInvocation Activation | Invoke the testing interface, whose URI is specified in the opaque expression of the OpaqueAction. Parse the result returned by the interface and emits the parsed result via its output pins. |
| | | ShellAction Activation | Execute a shell command or an executable shell file, whose location is specified in the opaque expression of the OpaqueAction. |

# Paper B

# Testing Self-Healing Cyber-Physical Systems under Uncertainty: A Fragility-Oriented Approach

Tao Ma, Shaukat Ali, Tao Yue, and Maged Elaasar

Software Quality Journal (SQJ). DOI: 10.1007/s11219-018-9437-3

# 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

# 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 [1]. 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"* [2]. 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 [3]. 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 [4]. For the state of art of testing CPSs, coverage-oriented structural testing is dominating [5]. 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 *fault*s: 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 [6]. 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) [7] 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 × three testing 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 [6]. The new contributions of this paper are: 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 2.5 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 (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, 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 [2], 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 $S_{sm_i} = \{s_{sm_i1}, \dots, s_{sm_ij}, \dots, s_{sm_is}\}$ 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 OCL[14] 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 $t \coloneqq (s_{src}, s_{tar}, op, g)$, 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[15] 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 fUML[16] and Precise Semantics Of UML State Machines (PSSM)[17] standards, the specified state machines are executable. Thus, the test model is called an ETM.

## 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 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 $\mathbb{T} = \{\mathbb{t}_1, \mathbb{t}_2, \dots, \mathbb{t}_\beta \dots \mathbb{t}_\mathrm{m}\}$. Each state $\mathbb{s}_\alpha$ 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 $[o_{sm_1x'} \wedge o_{sm_2y'} \wedge \dots \wedge o_{sm_nz'}]$ forms the state invariant of $\mathbb{s}_\alpha$, 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 [2]. 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 $\mathbb{t}_\beta$ 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

[14] http://www.omg.org/spec/OCL/2.4

[15] 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.

[16] http://www.omg.org/spec/FUML/1.2.1

[17] http://www.omg.org/spec/PSSM/1.0/Beta1

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 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 [2]. By executing the test model and the SUT at the same time, the framework can dynamically test the SUT against the model. Fig. 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.

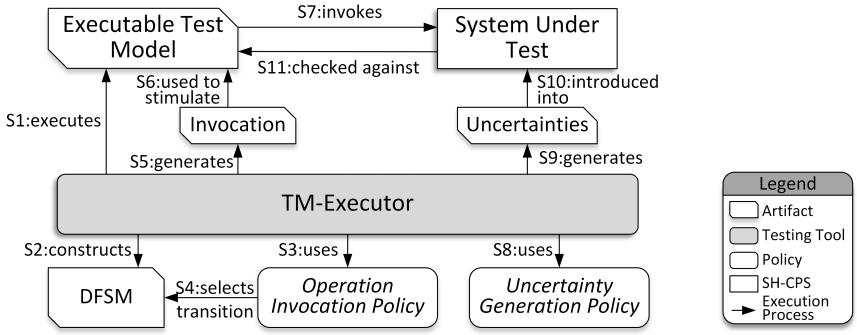


Fig. 1 Test Execution Process

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 approach to learning the optimal policy from interactions [8]. Consequently, we devise two reinforcement learning based algorithms to facilitate testing SH-CPSs under uncertainty.

Fig. 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.

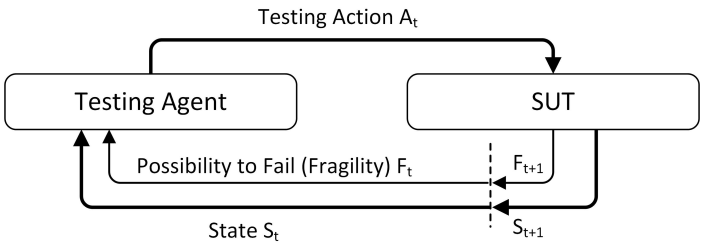


Fig. 2 Testing with Reinforcement Learning [8]

## 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) [9], which has been successfully applied together with reinforcement learning in many algorithms [10, 11].

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 an ANN with 100% accuracy [9], 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 [8].

In this paper, we divide the testing task into two sub-tasks. One is responsible for selecting operation invocations, and the other takes charge of introducing uncertainties. For the first sub-task, 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 sub-task is significantly larger than the one of the first sub-task. Due to this reason, we devise a tabular form based algorithm for the first sub-task and apply ANN to address the second sub-task.

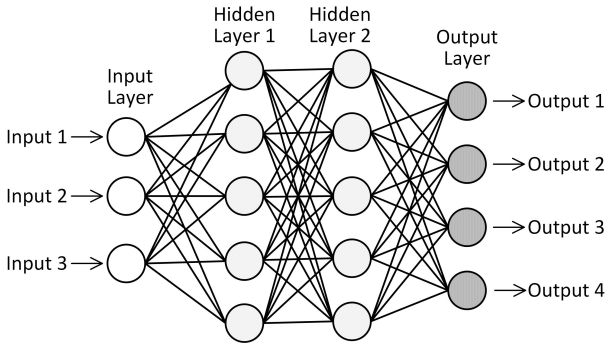


Fig. 3 Example of Artificial Neural Network

# 3 Running Example

We use an Unmanned Aerial Vehicle control system (ArduCopter[18]) as a running example to illustrate the problem of testing SH-CPS under uncertainty. Fig. 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 measurements bias from the GPS. The uncertainties are specified via the «Uncertainty» stereotype, provided in MoSH profile [2]. 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.

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*,

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

specify how the system should behave when an operation is invoked, and the self-healing behaviors specify how a fault is to be healed.

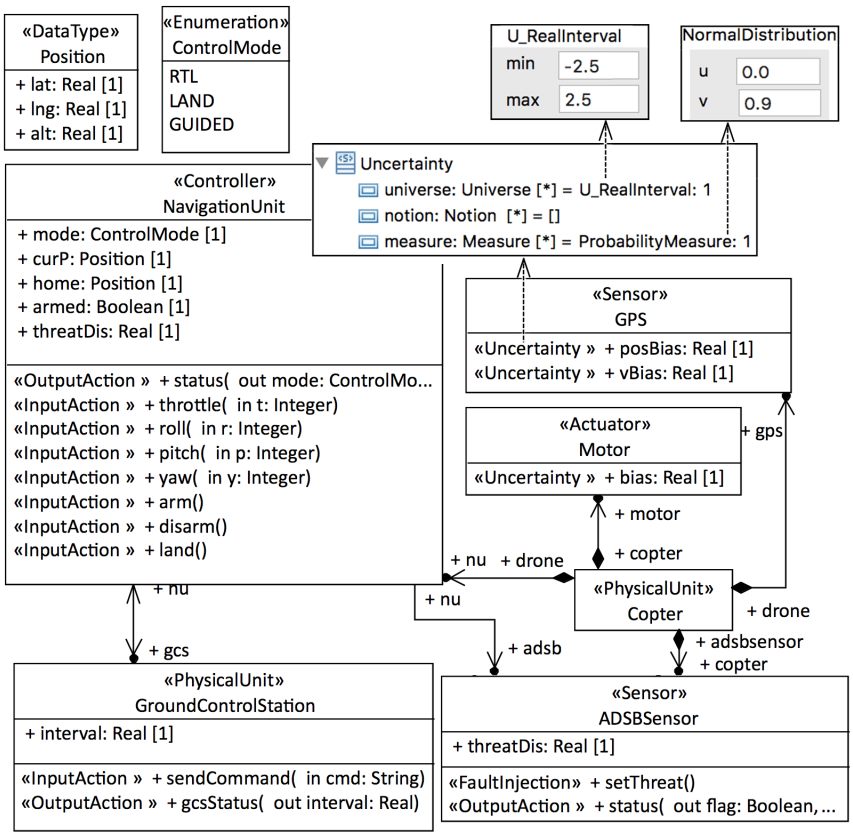


Fig. 4 Simplified Architecture of ArduCopter

*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.

Fig. 5 presents a partial simplified DFSM corresponding to the ETM for ArduCopter. We take one path (bold transitions in Fig. 4: $\mathbb{t}1 \rightarrow \mathbb{t}2 \rightarrow \mathbb{t}3 \rightarrow \mathbb{t}4 \rightarrow \mathbb{t}11 \rightarrow \mathbb{t}12 \rightarrow \mathbb{t}19 \rightarrow \mathbb{t}21$) 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 $\mathbb{t}2$ 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 [12]. 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 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 meters (not shown in Fig. 5), the collision threat is avoided and the *Copter*'s flight mode changes back to the previous one. Hence, 𝕥12 is traversed[19]. Then TM-Executor chooses to trigger 𝕥19, followed by firing 𝕥21, to stimulate the copter to pass through the *Landing* state and reaches the final state.

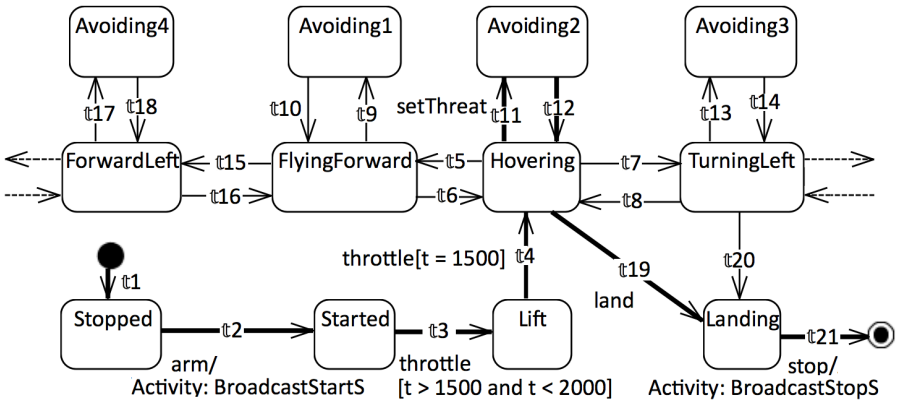


Fig. 5 Simplified Partial DFSM for ArduCopter

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 milliseconds. 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 milliseconds, 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

[19] When a collision is avoided, the copter is back to the flight mode. Hence, no testing interface 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.

active state's invariant, using the constraint evaluator DresdenOCL [13]. 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., $t17$ must be activated. However, activating $t17$ 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., $t3$ (Fig. 5). Each input leads to a distinct flight orientation and only in a few specific orientations, the collision is likely to 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.

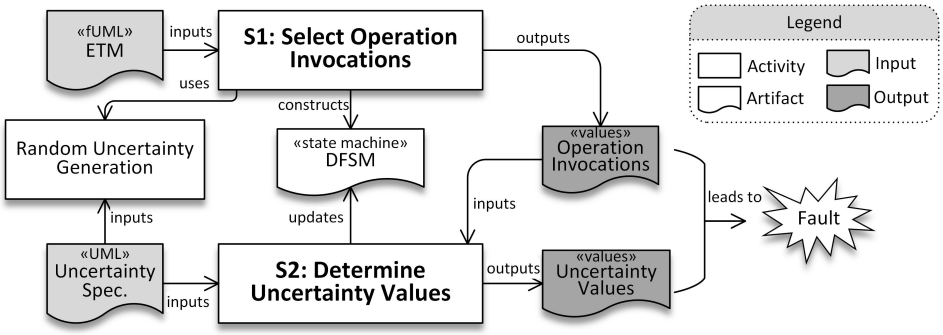


Fig. 6 Overview of Fragility-Oriented Testing under Uncertainty

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, 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 $\mathbb{s}$ is a real value between 0 and 1, denoted as $F(\mathbb{s})$. 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 [12]) 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 $dis\big(\neg(threatDis > 10)\big) = \frac{(15-10)+1}{(15-10)+1+1} = 0.86$, according to the distance function[20] defined in [12]. 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 $\mathbb{t}$. 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:

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

[20] 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 [12].

where $\gamma$ ($0 \leq \gamma < 1$) is a discount rate; $\mathbb{S}_{next}$ is a set of states that can be reached from $\mathbb{t}$'s target state via a path in the DFSM, and $n$ is the number of transitions between $\mathbb{s}$ and $\mathbb{t}$'s target state. As for testing, revealing faults via a short path is preferable, we penalize the fragility of a state by multiplying $\gamma^n$, if traversing at least $n$ transitions is required to reach the state from $\mathbb{t}$'s target state. For example, in Fig. 5, to obtain the T-value of $\mathbb{t}4$, we calculate the discounted fragility of the SUT in each state in $\mathbb{S}_{next}$. For the fragility corresponding to *Avoiding1*, it needs to be discounted by $\gamma^2$, since two transitions $\mathbb{t}5$ and $\mathbb{t}9$ have to be traversed to reach *Avoiding1* from $\mathbb{t}4$'s target state *Forward*. Clearly, the value of $\gamma$ 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 [8]. 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 $\mathbb{s}$, 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 (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.

If no invariant violation happens, the algorithm will evaluate the fragility of the SUT in $\mathbb{s}'$ (L15), i.e., $\mathbb{F}(\mathbb{s}')$, and use $\mathbb{F}(\mathbb{s}')$ to update estimated T-values. Since it is possible to reach $\mathbb{s}'$ via numerous transitions, finding all these transitions and 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 $\mathbb{t}11$ is invoked and the state of the SUT changes to *Avoiding2*, FOT evaluates the value of $\mathbb{F}(Avoiding2)$ and uses the value to update the T-value of $\mathbb{t}11$, i.e., $T(\mathbb{t}11)$. 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 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.

```
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)
       10         stateInvariant ← selectedTransition.target.invariant
       11         if executor.Evaluate(stateInvariant) is false
       12           LogFaultDetected(selectedTransition)
       13           dfsm.MarkFaultDetected(dfsm.activeState)
       14           break
       15         fragility ← 1- executor.DistanceToViolation(stateInvariant)
       16         executor.UpdateTvalue(selectedTransition, fragility) // revise the T-value of
                  selectedTransition
    End
```

## 5.2 T-value Learning

Before executing the SUT and the Executable Test Model (ETM), the T-value $T(\mathbb{t})$ of every transition is unknown. We adopt a reinforcement learning approach to learn $T(\mathbb{t})$

from execution. A fundamental property of $T(\mathbb{t})$ is that it satisfies a recursive relation, which is called the Bellman Equation [8], as shown in the formula below:

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

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 be 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 value 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 [14] 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.

### 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 [15] by assigning a selection probability to a transition proportional to the transition's T-value. The selection probability is given below (from [8]):

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

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 $\tau$ is a parameter, called temperature [16]. $\tau$ is a positive real value from 0 to infinity. A large $\tau$ causes transitions to be equally selected, whereas, a small $\tau$ 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 ($ET(\mathbb{t})$) 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 lead 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 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 $\mathbb{s}_i$ in the execution path corresponds to a number of state instances $\{\mathcal{s}_{i1}, \mathcal{s}_{i2}, \ldots, \mathcal{s}_{ik}\}$.

**Definition 3.** A state instance, $\mathcal{s}_{ij}$, is an instance of an abstract state, $\mathbb{s}_i$, 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., $\mathcal{s}_{ij} = \{v_1, v_2, \dots, v_{nsv}\}$. Based on the definition of fragility, we define the fragility of the SUT in a given state instance as follows:

$$F(\mathcal{s}_{ij}) = F(\mathbb{s}_i) = 1 - dis(\neg \mathbb{o}_i) \tag{7}$$

Note that although the state invariant $\mathbb{o}_i$ of a state $\mathbb{s}_i$ 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 $\mathcal{s}_{ij}$.

Fig. 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 one second to enter the next state *Started*. Within one second, 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, $\mathcal{s}_{ij+1}$, the SUT will switch to from the previous instance, $\mathcal{s}_{ij}$. 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.

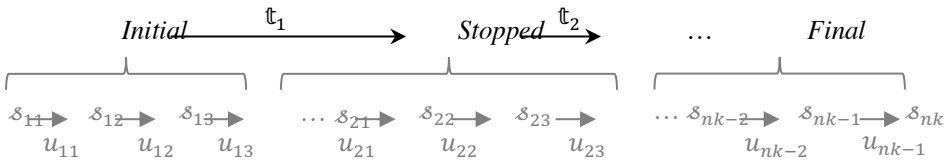


Fig. 7 Relations Between States and State Instances

## 6.1 Uncertainty Generation Policy

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

probability distribution of $u_{ij}$ given the condition that $s_{ij}$ is the current state instance. The conditional probability distribution can be changed, by adjusting policy parameters $\theta$.

As Artificial Neural Network (ANN) has been demonstrated to be an effective decision-making mechanism [17], 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 [18], 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 $\varepsilon$. By sampling from the distribution, we can decide the uncertainty values to be used for each state instance. Fig. 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.

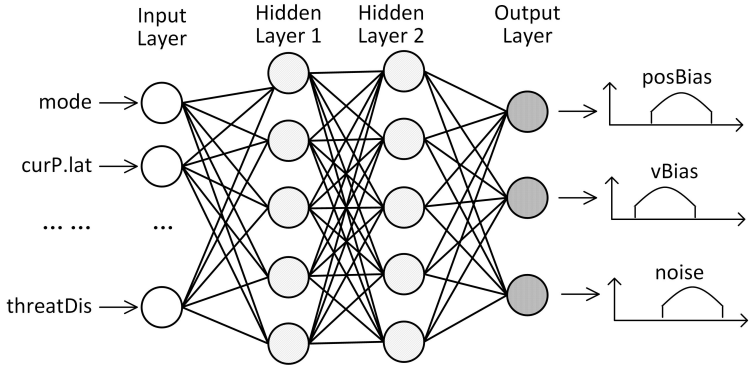


Fig. 8 Example of Uncertainty Generation Policy

## 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 $\mathcal{s}_{ij}$ 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 [18]:

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

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 $\pi_\theta$. $\rho_\theta$ 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 $\rho_\theta$ and $T_\theta(\mathcal{s}_{ij}, u_{ij})$ depend on the parameters of the policy.

Suppose we have two policies: $\pi_\theta$ 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 Equation (8).

To simplify the optimization problem, we choose to optimize an approximation of the goal function [19]:

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

Note that $\rho_\theta$ is changed to $\rho_{\theta'}$ and $T_\theta(s_{ij}, u_{ij})$ is changed to $T_{\theta'}(s_{ij}, u_{ij})$. This allows us to directly find the optimal improvement direction based on the $\rho_{\theta'}$ and $T_{\theta'}(s_{ij}, u_{ij})$ obtained from an existing policy $\pi_{\theta'}$, without extra executions. The general idea is that the value of $T_{\theta'}(s_{ij}, u_{ij})$ points out the expected reward of introducing uncertainty values $u_{ij}$ in a given state $s_{ij}$. To increase the total expectation, we just need to adjust the parameters of the policy to increase the probability to generate $u_{ij}$ in $s_{ij}$, if $T_{\theta'}(s_{ij}, u_{ij})$ is high. As proven in [18], as long as the Kullback–Leibler divergence, a distance measure, between the two policy $\pi_\theta$ 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 Equation (10), we replace the expectation over the uncertainty values following $\pi_\theta$ by the expectation over the uncertainty values following $\pi_{\theta'}$, according to importance sampling [20]:

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

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 $u_{11}$, $u_{12}$, … , $u_{nm}$ that leads to a high fragility, i.e., their $T_\theta(s_{ij}, u_{ij})$ is high, we adjust $\theta'$ to $\theta$ 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 [21] 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. Fig. 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 [22], 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 DresdenOCL [13]). 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 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 [12], 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.

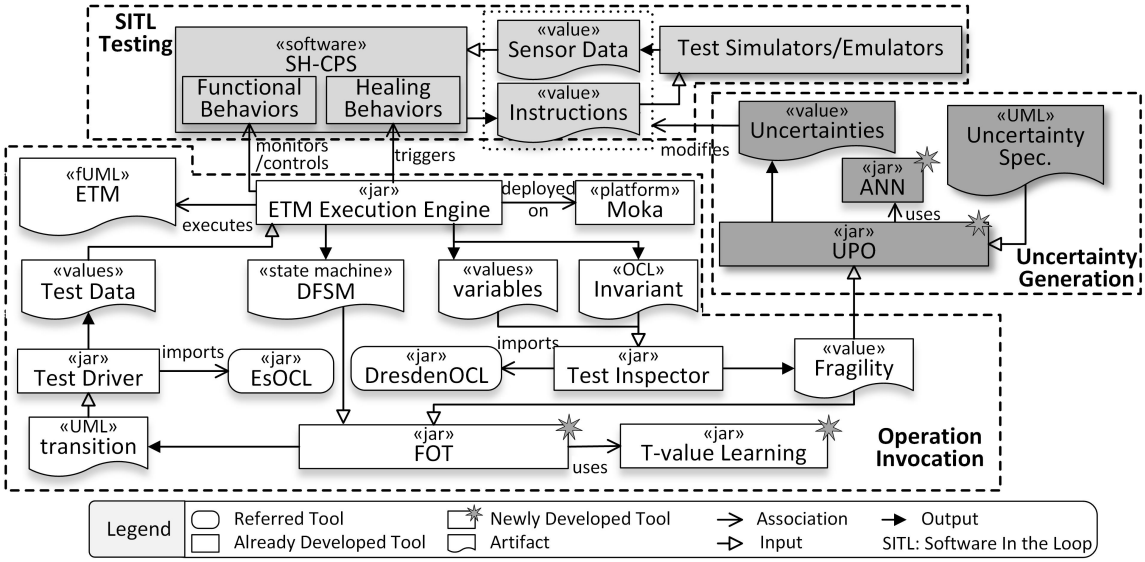


Fig. 9 SH-CPS Testing Framework [2]

# 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 [5]. 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, FOT and COT, for selecting operation invocations, and two algorithms, 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 ranges of uncertainties 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.

### 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) ArduRover[21] is an autopilot system for ground vehicles having two self-healing behaviors to avoid an obstacle and handle the disruption of control link; 3) ArduPlane[22] 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 15.

Table 15 Identified Uncertainties from the Three Case Studies

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

Before testing, we built the Executable Test Model (ETM) for each self-healing behavior of the three case studies. Table 16 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 16. Note that the average execution times are determined by the implementation of the SUTs, and they are not affected by different testing approaches.

Table 16 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 |

[21] http://ardupilot.org/rover/
[22] http://ardupilot.org/plane/

### 8.1.3 Experiment Tasks

Three tasks have to be performed to address the three research questions. Table 17 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 hours. 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%. 100% represents the standard variation ranges shown in Table 15. 80% (120%) means reducing (increasing) the ranges by 20 percent. For instance, the 80%, 100%, and 120% variation ranges of the uncertainty *Noise* from the accelerometer are (-7.2mg, +7.2mg), (-9mg, +9mg), and (-10.8mg, +10.8mg) 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.

Table 17 Overview of Experiment Design

<table>
<tr><th>RQ</th><th>Task</th><th>Metric</th><th>Statistical Test</th><th>Testing Approach</th><th>Testing Time(hour)</th><th>Scale of Uncertainty Variation</th><th>Case Studies</th></tr>
<tr><td>1</td><td>$T_1$: Compare the fault detection ability of the selected testing approaches</td><td>NDF</td><td>Mann-Whitney U test<br>Vargha and Delaney's $\hat{A}_{12}$</td><td rowspan="2">1: FOT+UPO<br>2: FOT+R<br>3: COT+UPO<br>4: COT+R</td><td rowspan="3">1: 72<br>2: 144<br>3: 216</td><td rowspan="3">1: 80%<br>2: 100%<br>3: 120%</td><td rowspan="3">1: ArduCopter (five SH Behaviors)<br>2: ArduPlane (two SH Behaviors)<br>3: ArduRover (two SH Behaviors)</td></tr>
<tr><td>2</td><td>$T_2$: Calculate the improvement with respect to the percentage of faults detected by the optimal approach</td><td rowspan="2">PDF</td><td>N/A</td></tr>
<tr><td>3</td><td>$T_3$: Analyze the correlations between the fault detection ability, testing time, and uncertainty scale</td><td>N/A</td><td>The optimal one</td></tr>
</table>

Consequently, 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 by the four approaches. Moreover, each testing job is performed 10 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.

### 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 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 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 [23], 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. 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 [2]. 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 $\gamma$ to 0.99 and temperature $\tau$ 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 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 $\varepsilon$ used by the stochastic policy is 0.4. These are commonly used settings in reinforcement learning [24]. The experiment is executed on Abel, a computer cluster at the University of Oslo[23]. Each testing job is run with eight cores and 32 GB RAM.

## 8.3 Experiment Results

**RQ1:** Table 18 presents the average number of faults detected by each testing approach (Task $T_1$). 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 19 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 hours. Thus, there is no significant difference among the four approaches.

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

Table 18 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 hours with 80% uncertainty range, S2: Test 144 hours with 80% uncertainty range, S3: Test 216 hours with 80% uncertainty range, S4: Test 72 hours with 100% uncertainty range, S5: Test 144 hours with 100% uncertainty range, S6: Test 216 hours with 100% uncertainty range, S7: Test 72 hours with 120% uncertainty range, S8: Test 144 hours with 120% uncertainty range, S9: Test 216 hours with 120% uncertainty range

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.

Table 19 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 | |
| | | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ | $\hat{A}_{12}$ | $p$ |
| 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 |
| | 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 |
| | 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 |
| S2 | FOT+R | **0.97** | **0.006** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.002** | **0.95** | **0.003** | **0.9** | **0.006** |
| | COT+UPO | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| | COT+R | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| S3 | FOT+R | **0.99** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.003** | **0.99** | **0.004** | **0.95** | **0.003** | **0.9** | **0.006** | **0.8** | **0.019** |
| | COT+UPO | **1** | **0.004** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.003** | **1** | **0.003** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| | COT+R | **1** | **0.004** | **1** | **0.004** | **1** | **0.004** | **1** | **0.003** | **1** | **0.003** | **1** | **0.003** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| S4 | FOT+R | **0.75** | **0.037** | **0.8** | **0.019** | **0.7** | **0.072** | **0.75** | **0.037** | **0.75** | **0.037** | **1** | **0.003** | **0.9** | **0.006** | **0.95** | **0.003** | **0.95** | **0.003** |
| | COT+UPO | **0.75** | **0.037** | **0.8** | **0.019** | **0.7** | **0.072** | **0.75** | **0.037** | **0.75** | **0.037** | **1** | **0.003** | **0.9** | **0.006** | **0.95** | **0.003** | **0.95** | **0.003** |
| | COT+R | **0.75** | **0.037** | **0.8** | **0.019** | **0.7** | **0.072** | **0.75** | **0.037** | **0.75** | **0.037** | **1** | **0.003** | **0.9** | **0.006** | **0.95** | **0.003** | **0.95** | **0.003** |
| S5 | FOT+R | **0.96** | **0.006** | **1** | **0.005** | **1** | **0.004** | **0.99** | **0.004** | **1** | **0.004** | **1** | **0.004** | **0.95** | **0.003** | **0.95** | **0.003** | **0.85** | **0.011** |
| | COT+UPO | **1** | **0.004** | **1** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.004** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| | COT+R | **1** | **0.004** | **1** | **0.005** | **1** | **0.004** | **1** | **0.004** | **1** | **0.004** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| S6 | FOT+R | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.005** | **1** | **0.004** | **0.95** | **0.003** | **0.85** | **0.011** | **0.75** | **0.037** |
| | COT+UPO | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.005** | **1** | **0.002** | **1** | **0.002** | **0.95** | **0.003** | **0.95** | **0.003** |
| | COT+R | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **1** | **0.005** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| 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** | **1** | **0.002** | **1** | **0.002** |
| | 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** | **1** | **0.002** | **1** | **0.002** |
| | 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** | **1** | **0.002** | **1** | **0.002** |
| S8 | FOT+R | **0.99** | **0.008** | **1** | **0.005** | **0.99** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.004** | **0.95** | **0.003** | **0.9** | **0.006** | **0.85** | **0.011** |
| | COT+UPO | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| | COT+R | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** | **1** | **0.002** |
| S9 | FOT+R | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **0.9** | **0.006** | **0.8** | **0.019** | **0.75** | **0.037** |
| | COT+UPO | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.003** | **1** | **0.002** | **0.95** | **0.003** | **0.95** | **0.003** |
| | COT+R | **1** | **0.004** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.005** | **1** | **0.002** | **1** | **0.002** | **0.95** | **0.003** | **1** | **0.002** |

**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 $T_2$). Table 20 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 hours to 216 hours. 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 hours to detect all the faults. Whereas, when *SU* is high, 144 hours are sufficient for the testing approach to find most faults.

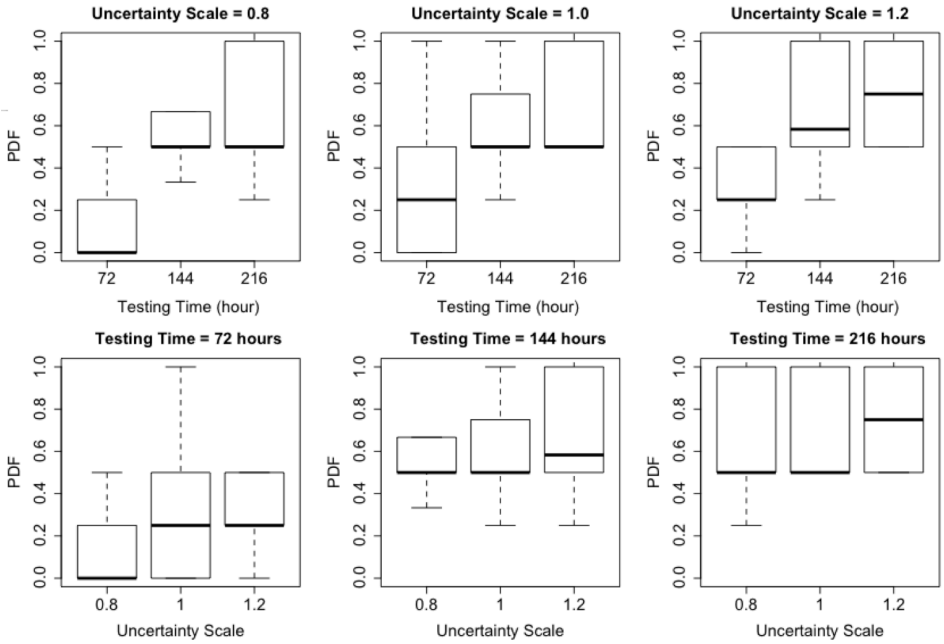


Fig. 10 Box Plots of PDF under Each Testing Time and Uncertainty Scale

Regarding the correlation between *PDF* and *SU*, it exposes similar phenomena. When *TT* is 72 hours, 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 hours, 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 *SU*s.

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

Table 20 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% |

## 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 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 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 [25]. 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 10 times. We applied 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 [25], 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 [25]. 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 [25] — 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 a 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 [26]. For a typical MBT approach, abstract test cases are generated from models first, e.g., using structural coverage criteria (e.g., all state coverage) [27, 28]. 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 [29, 30]. De Vries et al. [29] 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. [30] 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 [7]. 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. [31] 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 [32] 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, 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 [33, 34], and a conceptual model of uncertainty for CPSs has been built in [35]. To test systems in the presence of the uncertainty, Fredericks et al. [36] 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. [37] 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. [38] 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.

# Acknowledgement

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), Certus SFI and EU Horizon 2020 funded U-Test project (grant no. 645463).

# References

[1] T. Bures 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," ACM SIGSOFT Software Engineering Notes, vol. 40, no. 6, pp. 28-32, 2015.

[2] Ma, T., Ali, S., & Yue, T. (2019). Modeling foundations for executable model-based testing of self-healing cyber-physical systems. Software & Systems Modeling, 18(5), 2843-2873.

[3] S. Schupp et al., "Current challenges in the verification of hybrid systems," in International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems, 2015: Springer, 2015, pp. 8-24.

[4] X. Zheng and C. Julien, "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, 2015: IEEE Press, 2015, pp. 15-18.

[5] S. A. Asadollah, R. Inam, and H. Hansson, "A Survey on Testing for Cyber Physical System," in IFIP International Conference on Testing Software and Systems, 2015: Springer, 2015, pp. 194-207.

[6] T. Ma, S. Ali, T. Yue, and M. Elaasar, "Fragility-oriented testing with model execution and reinforcement learning," in IFIP International Conference on Testing Software and Systems, 2017: Springer, 2017, pp. 3-20.

[7] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," Formal Approaches to Software Testing and Runtime Verification, pp. 240-253, 2006.

[8] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction (no. 1). MIT press Cambridge, 1998.

[9] B. Yegnanarayana, Artificial neural networks. PHI Learning Pvt. Ltd., 2009.

[10] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," arXiv preprint arXiv:1708.05866, 2017.

[11] Y. Li, "Deep reinforcement learning: An overview," arXiv preprint arXiv:1701.07274, 2017.

[12] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating Test Data From OCL Constraints With Search Techniques," IEEE Transactions on Software Engineering, vol. 39, no. 10, pp. 1376-1402, 2013.

[13] 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, 2009: Citeseer, 2009, pp. 687-690.

[14] C. Z. Mooney, R. D. Duval, and R. Duval, Bootstrapping: A nonparametric approach to statistical inference (no. 94-95). Sage, 1993.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," Journal of artificial intelligence research, vol. 4, pp. 237-285, 1996.

[16] Y. Anzai, Pattern recognition and machine learning. Elsevier, 2012.

[17] J. E. Aronson, T.-P. Liang, and E. Turban, Decision support systems and intelligent systems. Pearson Prentice-Hall, 2005.

[18] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in Proceedings of the 32nd International Conference on Machine Learning (ICML-15), 2015: PMLR, 2015, pp. 1889-1897.

[19] S. Kakade and J. Langford, "Approximately optimal approximate reinforcement learning," in The Nineteenth International Conference on Machine Learning (ICML), 2002, vol. 2: PMLR, 2002, pp. 267-274.

[20] P. W. Glynn and D. L. Iglehart, "Importance sampling for stochastic simulations," Management Science, vol. 35, no. 11, pp. 1367-1392, 1989.

[21] M. R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems (no. 1). NBS, 1952.

[22] J. Tatibouet, "Moka – A simulation platform for Papyrus based on OMG specifications for executable UML," in EclipseCon, 2016: OSGI, 2016.

[23] 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), , 2011: IEEE, 2011, pp. 1-10.

[24] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in International Conference on Machine Learning, 2016, 2016, pp. 1329-1338.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering. Springer Science & Business Media, 2012.

[26] E. P. Enoiu, A. Cauevic, D. Sundmark, and P. Pettersson, "A Controlled Experiment in Testing of Safety-Critical Embedded Software," in 2016 IEEE International

Conference on Software Testing, Verification and Validation (ICST), 2016: IEEE, 2016, pp. 1-11.

[27] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297-312, 2012.

[28] W. Grieskamp, R. M. Hierons, and A. Pretschner, "Model-Based Testing in Practice," in Dagstuhl Seminar Proceedings, 2011: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[29] R. G. de Vries and J. Tretmans, "On-the-fly conformance testing using SPIN," International Journal on Software Tools for Technology Transfer (STTT), vol. 2, no. 4, pp. 382-393, 2000.

[30] 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, 2004: Springer, 2004, pp. 79-94.

[31] A. Groce et al., "Lightweight automated testing with adaptation-based programming," in IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE), 2012: IEEE, 2012, pp. 161-170.

[32] 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, 2016: ACM, 2016.

[33] A. J. Ramirez, A. C. Jensen, and B. H. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) 2012: IEEE, 2012, pp. 99-108.

[34] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in Software Engineering for Self-Adaptive Systems II: Springer, 2013, pp. 214-238.

[35] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model," in Modelling Foundations and Applications: 12th European Conference, ECMFA 2015: Springer 2015.

[36] E. M. Fredericks, B. DeVries, and B. H. Cheng, "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, 2014: ACM, 2014, pp. 17-26.

[37] W. Yang, C. Xu, Y. Liu, C. Cao, X. Ma, and J. Lu, "Verifying self-adaptive applications suffering uncertainty," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014: ACM, 2014, pp. 199-210.

[38] Zhang, M., Ali, S., & Yue, T. (2019). Uncertainty-wise test case generation and minimization for cyber-physical systems. Journal of Systems and Software, 153, 1-21.

# Appendix

Fig. 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.

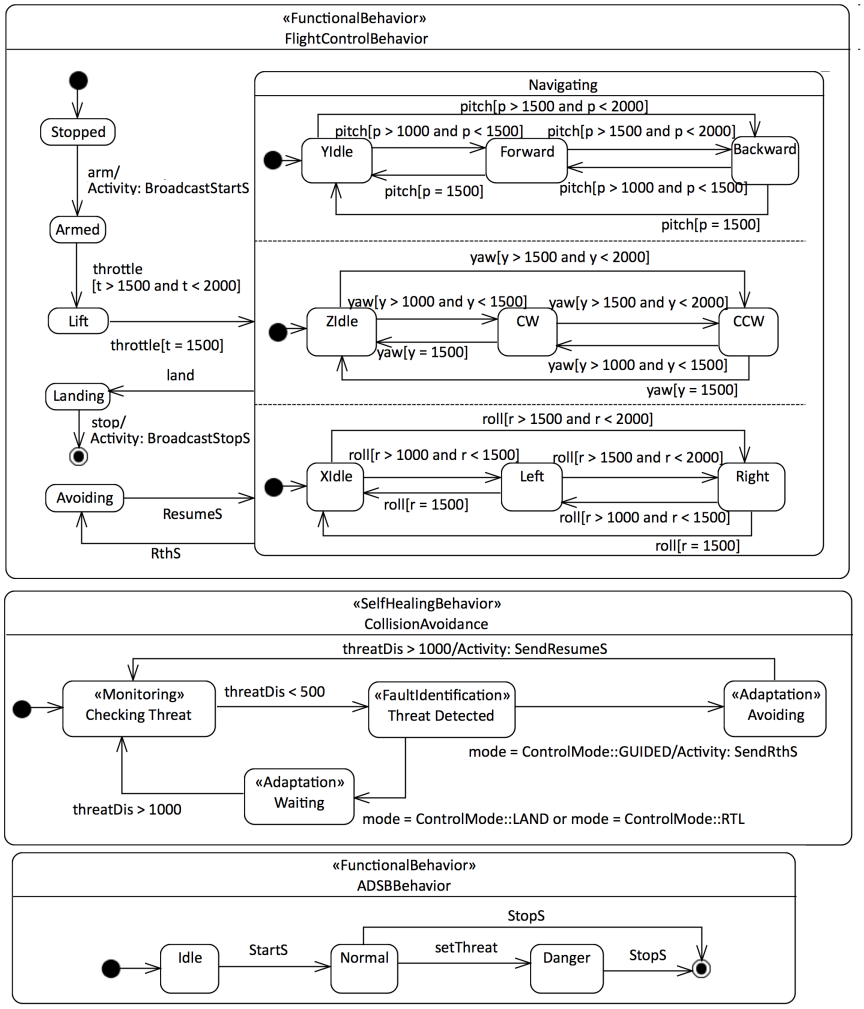


Fig. 11 Simplified ETM for ArduCopter

# Paper C

# Testing Self-Healing Cyber-Physical Systems under Uncertainty with Reinforcement Learning: An Empirical Study

Tao Ma, Shaukat Ali, and Tao Yue

Journal of Empirical Software Engineering (EMSE). DOI: 10.1007/s10664-021-09941-z

# Abstract

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

**Keywords** Cyber-Physical Systems, Uncertainty, Self-Healing, Model Execution, Reinforcement Learning, Empirical Evaluation

# 1 Introduction

As an essential feature of Cyber-Physical Systems (CPSs), self-healing enables a CPS to autonomously detect and recover from errors caused by software or hardware faults at runtime. We refer to this kind of CPSs as Self-Healing CPSs (SH-CPSs). Besides recovery, SH-CPSs have to address various uncertainties, which mean uncertain values that may affect behaviors of an SH-CPS during execution, including measurement errors from sensors and actuation deviations from actuators. In reality, uncertainties are uncontrollable and exact values of errors are unknown for a given interaction between an SH-CPS and its environment. To assess the reliability of SH-CPSs, we would like to test if an SH-CPS can still behave as expected under uncertainties, with the range of each uncertainty given. As SH-CPSs have two kinds of behaviors (i.e., functional behaviors for fulfilling business requirements and self-healing behaviors for handling faults [1]) both affected by uncertainties, we aim to test both kinds of behaviors of SH-CPSs under uncertainties.

To solve the testing problem, previously, we proposed a fragility-oriented approach [2]. In this approach, we evaluate how likely the system will fail in a given state, defined as fragility, and use the fragility as a heuristic to find the optimal testing policies for detecting failures, i.e., unexpected behaviors. Here, we need to find two policies. The first policy is used to decide how to exercise the SH-CPS by invoking its testing interfaces. Meanwhile, another policy is used to determine the value of each uncertainty that affects a measurement or actuation when an SH-CPS uses a sensor or actuator to monitor or change its environment. The value is then passed to simulators of sensors or actuators to replicate the uncertainty's effect. In our previous work [2], reinforcement learning has demonstrated its effectiveness in learning these two policies. Compared with random testing and a coverage-oriented testing approach, the fragility-oriented approach with reinforcement learning discovered significantly more failures. However, as several reinforcement learning algorithms are available [3], there is no sufficient evidence showing which reinforcement learning algorithms are the best to be used for testing SH-CPS under uncertainties.

To this end, we conducted this empirical study, in which the performance of 14 combinations of various reinforcement learning algorithms was evaluated together with the fragility-oriented approach for testing six SH-CPSs. As aforementioned, to detect failures,

the algorithms need to learn the optimal policy for invoking testing interfaces (task 1) and learn the best strategy to choose uncertainty values (task 2). As these two tasks are different, we applied two sets of algorithms to perform them. Specifically, we applied two value function learning based algorithms, Action-Reward-State-Action (SARSA) [4] and Q-learning [4], for finding the policy of invoking testing interfaces, and seven policy optimization based algorithms, Asynchronous Advantage Actor-Critic (A3C) [5], Actor-Critic method with Experience Replay (ACER) [6], Proximal Policy Optimization (PPO) [7], Trust Region Policy Optimization (TRPO) [8], Actor-Critic method using Kronecker-factored Trust Region (ACKTR) [9], Deep Deterministic Policy Gradient (DDPG) [10], and Uncertainty Policy Optimization (UPO) [2], for learning the policy of selecting values for uncertainties.

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

The remainder of this paper is organized as follows. Section 2 provides background information, followed by the experiment design in Section 3. Section 0 and Section 5 present the experiment execution and results, with a discussion about the results and alternative approaches given in Section 6. Section 7 analyzes threats to validity. After a discussion about related work in Section 8, Section 9 concludes the paper.

# 2 Background

This section briefly introduces the test model used to capture components, expected behaviors, and uncertainties of the SH-CPS under test in Section 2.1. Section 2.2 explains how to test the SH-CPS against a test model. The problem is formulated in Section 2.3, and

Section 2.4 shows the reinforcement learning algorithms that can be used to solve the testing problem. In this section, key concepts related to testing and reinforcement learning are italicized.

## 2.1 Uncertainty-Wise Executable Test Model

To test SH-CPSs under uncertainty, in our previous work [1], we have proposed the *Executable Model-Based Testing approach* (*EMBT*). In this approach, an *executable test model* is used to capture components, uncertainties, and expected behaviors of an SH-CPS under test. This section will use an example of an autonomous Copter system to introduce the *test model*, a simplified which is given in Fig. 12.

An *executable test model* consists of a collection of UML[24] class diagrams and state machines. The class diagrams capture components of the SH-CPS under test as UML classes. The components' state variables that are accessible via testing interfaces are specified as properties of the classes and the testing interfaces for controlling or monitoring the components are defined as operations. For example, the Copter has a NavigationUnit and a GPS, and they are specified as UML classes (Fig. 12 (A)). The properties of the NavigationUnit capture its state variables, like "velocity" and "height", that can be queried by the testing interface status(). Besides status(), the NavigationUnit provides several interfaces used to control the flight, including throttle(), pitch(), arm(), and land(). They are all specified as operations of the NavigationUnit. When an SH-CPS uses sensors and actuators to monitor and change its environment, the measurement and actuation performed by the sensors and actuators are affected by uncertainties. Such uncertainties are specified as stereotyped[25] properties, with their ranges specified as stereotype attributes. For instance, the measurement error of " velocity" measured by GPS is an uncertainty. It is specified as a property of the GPS class, "vError", and its range is specified by the stereotype attributes: "min" and "max". Note that as the probability distributions of the uncertainty may not be known, our testing approach does not test SH-CPSs with uncertainties sampled from distributions. Instead, it applies effective algorithms to find the

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

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

values of uncertainties that can prohibit a system from behaving normally. Therefore, only the valid range of each uncertainty is defined in the test model.

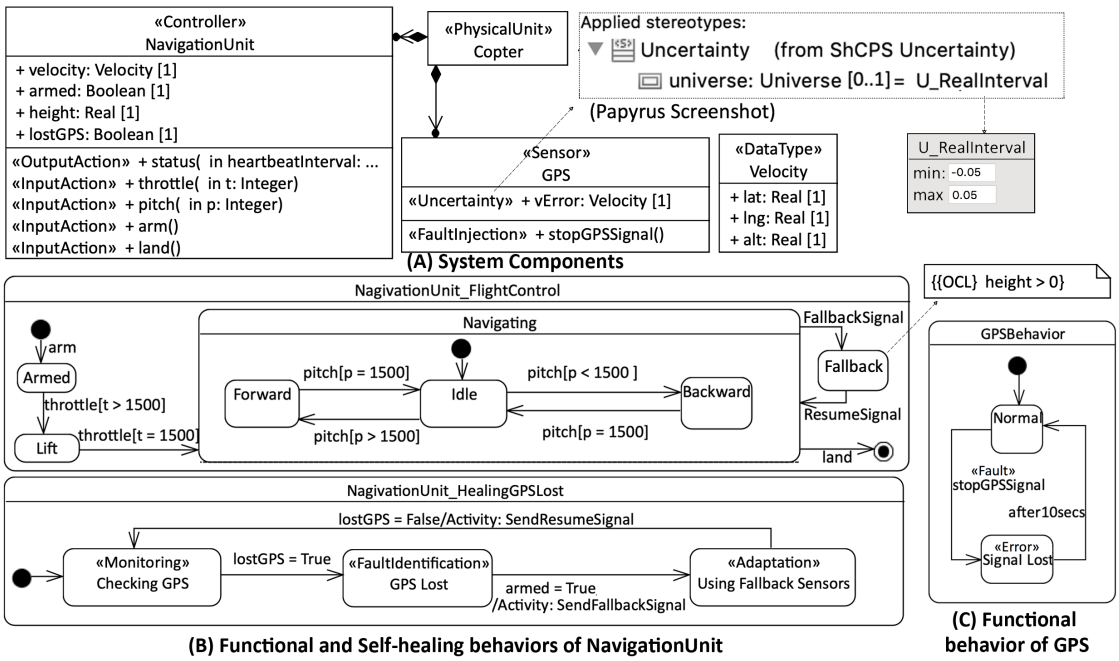


Fig. 12 Simplified Test Model of a Copter System

Expected behaviors of each component are specified as UML state machines. A state in a state machine is defined together with a state invariant, which is an OCL[26] constraint of state variables. By evaluating the invariant, with the variables' values accessed from the system under test, we can check if the invariant is satisfied when a component is supposed to be in a given state. For example, Fig. 12 (B) presents two expected behaviors of the NavigationUnit. The first behavior specifies how the NavigationUnit controls a flight in response to invocations of its operations. For example, the Copter starts to take off when throttle() is invoked with its parameter "t" above 1500. Normally, the NavigationUnit employs the GPS to monitor the flight. When the GPS loses its signal and fails to measure the Copter's position and velocity, a self-healing behavior (i.e., an adaptive control algorithm) will detect the incorrect measurement, identify the cause (i.e., GPS fault) and switch to other sensors[27] until the GPS outage is over. During this period, the self-healing behavior controls the Copter's movement properly to avoid it crashing on the ground. The second state machine in Fig. 12 (B) specifies this self-healing behavior. The state invariant

[26] OCL: Object Constraint Language. https://www.omg.org/spec/OCL/
[27] Other sensors include barometer, accelerometer and gyroscope. Limited by the space, they are not shown in Fig. 1.

of “Fallback” requires that the Copter should be above the ground, i.e., “height > 0”, when the behavior takes effect.

Because the self-healing behavior is internal, it is controlled by the NavigationUnit, instead of external instructions. Consequently, it needs a fault injection operation (e.g., stopGPSSignal() defined in the GPS class ) to trigger such behavior. We also use UML state machines to specify when a fault can be injected and how it will affect the state of a corresponding component, with the stereotypes of «Fault» and «Error» provided by our modeling framework [2]. For instance, Fig. 12 (C) presents the behavior of GPS. Initially, GPS is in the “Normal” state. Then, stopGPSSignal() can be invoked, and it will trigger GPS switching to the “Signal  Lost” state, in which the GPS will stop measuring position and velocity to mimic the fault of losing signal from GPS. After 10 seconds, GPS will switch back to the “Normal” state and start measuring again. The state machine tells us that this is a transient fault. In contrast, a permanent fault will keep a component in an error state. Based on the UML state machines, an algorithm can inject the fault in various conditions and learn when a fault should be injected to reveal an unexpected behavior.

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

## 2.2 Uncertainty-Wise Executable Model-Based Testing

To efficiently test SH-CPSs, we proposed an executable model-based testing approach. In this approach, an SH-CPS is tested against a test model, by executing the system and the model together, sending them the same testing stimuli (e.g., *operation invocation*s) and comparing their consequent states. To realize the approach, we have developed a testing framework. In this subsection, we briefly introduce the theoretical foundations of the executable model-based testing approach in Section 2.2.1 and then present the testing framework in Section 2.2.2. More details about the theoretical foundations and implementation can be found in our previous work [1].

### 2.2.1 Theoretical Foundations

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

First, the standard of Semantics of a Foundational Subset for Executable UML Models (fUML) [11], Precise Semantics of UML State Machines (PSSM) [12], and our extensions [1] provide precise execution semantics of UML model elements that are used to specify a test model. The semantics enables the test model to be executed in a deterministic manner.

Second, the Object Constraint Language (OCL) [13] standard gives us a standard way to specify constraints that an SH-CPS has to satisfy during execution. As explained in the previous section, each state in a test model is defined together with a state invariant, i.e., a constraint on the values of state variables in OCL. By evaluating the invariant with actual values of the state variables obtained from the system under test, we can rigorously check if the system behaves as expected in a given state.

Third, the Functional Mockup Interface (FMI) standard has defined a way to co-execute hybrid models. Based on the standard, we have devised a co-execution algorithm and implemented a testing framework [1] to enable a test model to be executed together with an SH-CPS, even though the test model and components of the SH-CPS are implemented with diverse modeling paradigms.

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

### 2.2.2 Implementation (TM-Executor)

To realize the executable model-based testing, we have developed a testing framework, TM-Executor. Fig. 13 presents an overview of the framework. As shown in the figure, a *test model* is executed by an Execution Engine, together with the SH-CPS under test. To drive the execution under uncertainties, a Test Driver has to invoke operations on the system and the model to control their behaviors. Meanwhile, an Uncertainty Introducer needs to introduce uncertainties in the environment to replicate the effects of measurement errors and actuation deviations via simulators of sensors and actuators. The two parallel processes determine how an SH-CPS is tested under uncertainties.

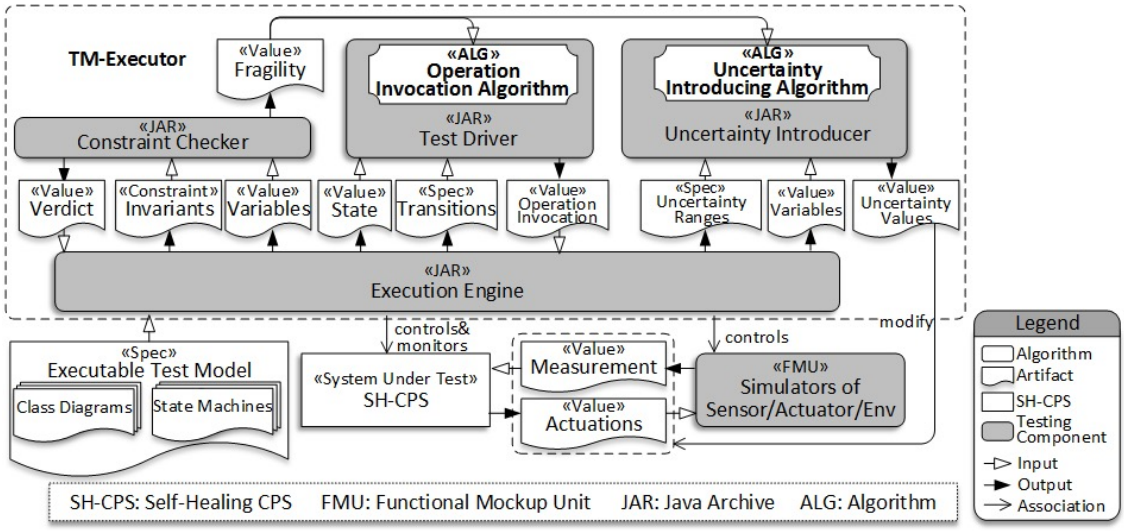


Fig. 13 Overview of the TM-Executor Testing Framework

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

**Definition 1.** An *operation invocation* is a combination of an operation and its input parameter values that are used to call the operation and trigger a transition defined in a *test model*.

For instance, when the active states of the NavigationUnit and GPS are (“Idle”, “Checking GPS”, “Normal”) as shown in Fig. 12, the Test Driver needs to choose either to invoke pitch() to let the NavigationUnit switch to state “Forward” or “Backward”, or to call stopGPSSignal() to inject a GPS fault. When an operation is selected, and an *operation invocation* is generated by the Test Driver, the Execution Engine will perform the

invocation to trigger an outgoing transition in the test model. Meanwhile, the Execution Engine invokes the testing interface represented by the operation with the same input parameter values, to make the system enter the target state of the outgoing transition as well. To check if the consequent states of the SH-CPS and *test model* are the same, the Execution Engine obtains state variables' values from the SH-CPS via testing interfaces, and passes the values to a Constraint Checker[28] to evaluate the invariant of the target state. If the invariant is not satisfied, it means that the SH-CPS failed to behave consistently with the *test model*, and thus a failure is revealed. Otherwise, the Test Driver will keep on generating *operation invocation*s to drive the execution until a terminal state is reached.

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

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

The *uncertainty value* is used to modify measurements performed by the sensors and actions performed by actuators to simulate the effect of uncertainties. For example, an *uncertainty value* is chosen for the measurement error of GPS, "vError" (shown in Fig. 12 (A)), for each velocity measured by the GPS. By adding the measurement error ("vError") to the true value of the velocity, derived from a simulation model, we can replicate the effect of the uncertainty, and test if the system will violate any invariants with the selected measurement errors.

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

## 2.3 Problem Formulation

As explained in the previous section, testing an SH-CPS under uncertainties involves two parallel processes: 1) invoking operations on the system to explore its behaviors and 2)

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

introducing uncertainties in the environment to simulate the effects of measurement errors and actuation deviations. The two processes are independent from each other, since in the real-world the uncertainties keep changing, independent of operation invocations performed on the system.

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

**Task 1.** Given a *test model*, find the optimal sequence of *operation invocation*s to maximize the chance of detecting failures, with uncertainties uniformly sampled from their ranges.

**Task 2.** Given a *test model* and a sequence of *operation invocation*s, find the sequence of *uncertainty value*s that makes the SH-CPS under test violate an invariant during handling the *operation invocation*s.

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

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

$$F(s_t) = \ 1 - dis(\neg o_t, V) \quad (1)$$

where $o_t$ is a state invariant, i.e., a constraint on the values of a set of state variables in OCL (Section 2.1), $V$ is the values of state variables, $dis(\neg o_t, V)$ is a distance function that returns a value between 0 and 1 indicating how close the constraint $\neg o_t$ is to be satisfied by $V$. The distance functions for all types of OCL constraints can be found in [14].

Take the Copter shown in Fig. 12 as an example. Assume the Copter is in the "Fallback" state. Its state invariant is "height > *0*", where height is a state variable, representing the distance between the Copter and ground. This invariant requires that height must be larger than zero to avoid crashing on the ground (i.e., "height=0" means crashing). If height equals to 10, the *fragility* of the Copter equals to:$1 - dis[\neg(height \ > 0)] = 1 - \frac{10}{10+1} = 0.09$, according to the distance function given in [14]. If the height is reduced to 1, then the *fragility* will be increased to 0.5, indicating the Copter is closer to crash on the ground. When height is reduced to zero, and the invariant is violated, the *fragility* is increased to one, its maximum value.

In the context of testing, the purpose of the *agent*s is to discover failures, and thus they are interested in finding *action*s that can lead to a violation of an invariant, i.e., making *fragility* equal to 1, rather than increasing the sum of *fragilities*. For instance, one sequence of *action*s makes an SH-CPS go through three states: $s_1$, $s_2$, and $s_3$, with their *fragilities* being 0.0, 0.0, and 0.9, respectively. Another sequence leads to states $s'_1$, $s'_2$, $s'_3$, $s'_4$, and $s'_5$ with their fragilities being 0.1, 0.2, 0.1, 0.3, and 0.3, respectively. Though the later sequence of *action*s obtains a higher sum of *fragilities*, it is less likely to detect a failure than the first sequence. Therefore, we adapt the objective of reinforcement learning from maximizing cumulative *reward*s to maximizing a future *reward*, i.e., increasing the maximum *fragility* that can be reached in the future, as defined below:

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

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

objective, we also need to update two value functions that are broadly used by reinforcement learning algorithms, as discussed below.

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

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

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

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

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

Based on the adapted objective, we can apply reinforcement learning algorithms to address the two tasks sequentially. For Task 1, an algorithm takes the current active state and its outgoing transitions as inputs. As a state only has finite outgoing transitions, the input space of the algorithm is finite and discrete. The output of the algorithm is one of the outgoing transitions. A test data generator, EsOCL [14], is used to generate an *operation invocation* (including an operation and valid inputs of the operation) to activate the trigger specified on the transition. Consequently, the algorithm only needs to choose one of the outgoing transitions as output, and thus its output space is also discrete. After a number of episodes[29], the sequence of *operation invocation*s that leads to the highest *fragility* is chosen to be the optimal one. It is used in Task 2 to find the uncertainty values that can reveal a system failure. For the algorithms used to address Task 2, their inputs are the ranges of *uncertainty value*s and the *state* of the SH-CPS under test, reified as state variables' values of the system, like the velocity and position of the Copter. The output of the algorithm is an *uncertainty value* for each uncertainty. As the state variables' values and *uncertainty value*s are continuous, both input and output spaces of the algorithm are continuous. As the two tasks have different characteristics, they pose different

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

requirements for reinforcement learning algorithms. The following section presents the state-of-the-art algorithms for solving the tasks.

## 2.4 Reinforcement Learning Algorithms

In general, reinforcement learning algorithms can be classified into value function learning based approaches and policy optimization based approaches [3]. Based on these two categories and more detailed subcategories proposed in literature reviews [3, 15, 16] we selected a benchmark reinforcement learning algorithm for each subcategory, summarized in Table 22 and Table 21. More details are given in the following subsections.

### 2.4.1 Value Function Learning Methods

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

**State-Action-Reward-State-Action (SARSA)** [4] SARSA uses the following equation to learn $Q_\pi$ for policy $\pi$.

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

where $Q_\pi$ is estimated *Q function* for $\pi$; $\alpha$ is a learning rate, which controls the step size of each update; $Err_{\pi,t}$ is the *temporal difference error*. Based on the adapted objective of reinforcement learning (Section 2.3), $Err_{\pi,t}$ is calculated by $max\,[F(s_t), \gamma \cdot Q_\pi(s_{t+1}, a_{t+1})] - Q_\pi(s_t, a_t)$. When a sample of (*state*, *action*, *reward*) is obtained, SARSA takes Equation (5) to update $Q_\pi(s_t, a_t)$. To collect the sample, SARSA applies the ε-greedy policy to select *action*s. That is, with a probability of ε, the policy randomly selects from all possible *action*s, and with a probability of 1- ε, it selects the *action* with the highest Q value. In theory, SARSA can converge to the *Q function* of the optimal policy, as long as all *state-action* pairs are visited an infinite number of times, and the ε-greedy converges in the limit to the greedy policy, i.e., reducing ε to zero [17].

**Q-learning** [4] Instead of learning the *Q function* of a given policy as SARSA does, Q-learning tries to learn the *Q function* of the optimal policy directly, independent of the policy being followed:

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

where $Q_*(s_t, a_t)$ represents the estimated *Q function* of the optimal policy, and $Err_{*,t}$ is the temporal difference error for $Q_*$, calculated by $max\,[F(s_t), \gamma \cdot Q_*(s_{t+1}, a_{t+1})] - Q_*(s_t, a_t)$. In this case, the policy, $\pi$, used by Q-learning only determines which *state-action* pair is to be visited, while the *state-action* pair and observed *reward* are used to update $Q_*$, rather than $Q_\pi$. As all pairs of *state-action* continue to be visited, Q-learning will gradually learn the *Q function* and find the corresponding optimal policy [4].

Table 21 Policy Optimization Based Reinforcement Learning Algorithms*

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

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

Table 22 Value Function Learning Based Reinforcement Learning Algorithms*

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

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

### 2.4.2 Policy Optimization Methods

In contrast to value function based methods, policy-based reinforcement learning algorithms maintain a *policy network* (an Artificial Neural Network (ANN)) to select *action*s. The *policy network* takes the *state* of the *environment* as input and outputs an *action* that is to be performed on the *environment*. By following the *policy network*, the policy-based algorithms collect samples of (*state*, *action*, *reward*), optionally take the

samples to evaluate the policy determined by the *policy network*, and then optimize the *policy network* based on the samples or evaluation. The main differences among the policy-based algorithms lay in the method of policy evaluation, optimization, and whether/how samples can be reused for the evaluation.

**Asynchronous Advantage Actor-Critic (A3C)** [5] In A3C, multiple threats are run in parallel to collect samples of (*state*, *action*, *reward*) by following a threat dependent *policy network*. The samples are used to train a *critic* (another ANN) that estimates an *advantage function* for evaluating the policy. The *advantage function*, $A_\pi(s_t, a_t)$, equals to $Q_\pi(s_t, a_t) - V_\pi(s_t)$. It reveals how good an *action* $a_t$ is to be taken in a given *state* $s_t$, compared with the average value of all candidate *action*s in *state* $s_t$. Each threat updates its *policy network* by the gradient of the "goodness" of *action*s with respect to the parameters of its policy, i.e., $\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A_\pi(s_t, a_t)$. From time to time, the local *policy network*s are synced with a global one, so that the threats can work together to learn the optimal policy.

**Actor-Critic method using Kronecker-factored Trust Region (ACKTR)** [9] Compared with the gradient taken by A3C, a *natural gradient* can give a better direction for improvement. However, computing *natural gradient* is extremely expensive. To reduce the computational complexity, ACKTR proposes to use Kronecker-Factored Approximated Curvature (K-FAC) [18] to obtain an approximate *natural gradient* and take the approximate gradient to optimize the *policy network* and *critic*.

**Deep Deterministic Policy Gradient (DDPG)** [10] From another perspective, DDPG uses *Q function* instead of the *advantage function* to evaluate the goodness of *action*s. It calculates the gradient of *Q function* with respect to the policy's parameters, $\nabla_{a_t} Q(s_t, a_t) \cdot \nabla_\theta \pi_\theta(a_t|s_t)$, and uses the gradient to update the *policy network* to make the network select *action*s with the highest *Q value*.

**Trust Region Policy Optimization (TRPO)** [8] A shortcoming of aforementioned algorithms is that they need to recollect samples of (*state*, *action*, *reward*) to evaluate the *policy network* after each update. To improve the sample efficiency, *importance sampling* can be used as an off-policy estimator to estimate the *advantage function* or *Q function* of a given policy, using samples collected under other policies:

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

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

where $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is called *importance weight*. However, if $\pi_\theta(a_t|s_t)$ deviates too much from $\pi_{\theta_{old}}(a_t|s_t)$, importance sampling will have high variance. Imagine that, if $\pi_{\theta_{old}}(a_t|s_t)$ is zero for a pair of $(s_t, a_t)$ and $\pi_\theta(a_t|s_t)$ is greater than zero, the *importance weight* will become infinite. Therefore, to use importance sampling, it is necessary to bound the difference between the two policies. To do so, TRPO adds a *KL divergence* [19] constraint to each policy update, and it transforms the reinforcement learning problem into a constrained optimization problem:

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

where $\rho_{\theta_{old}}$ is the *state* distribution determined by policy $\pi_{\theta_{old}}$; $Q_{\theta_{old}}$ is the *Q function* of policy $\pi_{\theta_{old}}$; $\overline{D_{KL}}(\theta_{old}|\theta)$ is average *KL divergence* between two policies; and $\delta$ controls the maximum step size for one policy update. The KL divergence constraint defines a trust region for a policy update. When the constraint is met, TRPO can guarantee a monotonic improvement for the *policy network*. To efficiently solve the constrained optimization problem, TRPO applies the Conjugate Gradient Algorithm [20] to approximately calculate *natural gradient* and follows the direction of the gradient to find the solution of the optimization problem.

**Proximal Policy Optimization (PPO)** [7] As TRPO is relatively complicated, PPO was proposed to use a clip function as an alternative to the *KL divergence* constraint. The clip function is:

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

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

**Actor-Critic method with Experience Replay (ACER)** [6] To improve sample efficiency and stabilize the estimation of the value function, ACER was proposed with three techniques. First, it uses *Retrace* [21] to estimate the *Q function* of the current policy, using samples collected under past policies. As proven in [21], *Retrace* has low variance, and it can converge to the *Q function* of a given policy, using samples collected from any policies.

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

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

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

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

**Uncertainty Policy Optimization (UPO)** [2] Different from the aforementioned methods, which apply a *critic* to evaluate their policy, UPO directly searches the space of all possible policies to find the optimal one. In UPO, the policy is decomposed into a probability distribution and a *policy network* that outputs statistics of the distribution. For example, if we choose to use the normal distribution, the outputs of the *policy network* will

be mean and variance of the distribution. Therefore, given a distribution, the *policy network* determines the policy used to select *action*s. In the beginning, UPO starts with a randomly initialized *policy network*, and it keeps on selecting *action*s by following the policy determined by the network. When UPO observes a sequence of *action*s leading to a higher *fragility*, it calculates the conjugate gradient [20] of the *policy network*, multiplied with $Q$ value, i.e., $\nabla_\theta^{con} \log \pi_\theta(a_t|s_t) \cdot Q_{best}(s_t, a_t)$, where $\nabla_\theta^{con} \log \pi_\theta(a_t|s_t)$ is the conjugate gradient and $Q_{best}(s_t, a_t)$ is the $Q$ *value* of a pair of *state* and *action* that has been observed by performing the sequence of *action*s. Afterward, UPO takes the gradient to update parameters of the *policy network*, so as to increase the selection probability of this sequence of *action*s. UPO continues the search process until reaching the maximum iterations.

# 3 Experiment Planning

Following the guidelines of conducting and reporting empirical studies [22] [23] [24], we designed and conducted the experiment. Section 3.1 presents our research goals. Sections 3.2 to 3.4 describe the rationale of choosing candidate algorithms, subject systems, and testing tasks performed by the algorithms. Hypotheses and related variables are described in Section 3.5, and Section 0 explains the applied statistical tests.

## 3.1 Goals

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

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

**Goal 2.** In the context of uncertainty-wise executable model-based testing, analyze the scalability of the reinforcement learning algorithms to examine their abilities to be applied for testing complex SH-CPSs.

## 3.2 Algorithms under Investigation

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

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

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

In summary, two reinforcement learning algorithms were used for selecting operation invocations, and seven algorithms were applied to choose uncertainty values. In total, there are 14 combinations of the algorithms, also called 14 testing approaches (denoted as *APP*)

in the following. We implemented SARSA, Q-learning, and UPO by ourselves, and the other algorithms were taken from OpenAI Baselines[30].

## 3.3 Subject Systems

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

### 3.3.1 System Description

ArduCopter (AC)[31] is a full-featured, open-source control system for multicopters, helicopters, and other motor vehicles. It can cater to a range of flight requirements, from First Person View racing, to aerial photography and autonomous cruising. It is equipped with five **self-healing behaviors**. Two of them are rule-based policies used to detect the disconnection between a copter and its radio controller or ground control station, and guide the copter to return and land. Another self-healing behavior is a quantitative model-based method [25] for detecting measurement errors caused by a transient GPS fault. When such an error is detected, the behavior will identify the fault and adapt the copter to use other sensors. The other two self-healing behaviors are two control algorithms that are used in the event of high vibration, e.g., strong wind, to stabilize the flight, and used to avoid collision with an intruding aerial vehicle. In total, AC uses four sensors (one GPS, one accelerometer, one gyroscope, and one barometer) and one actuator (a motor) to monitor and control the flight. Table 23 shows the eight types of uncertainty related to these sensors and the actuator. The ranges of the uncertainties are specified in their product specification. Each type of uncertainty affects measurement errors or actuation deviations in three dimensions, i.e., longitude, latitude, and altitude of the copter. Therefore, there are three instances for each uncertainty type. In total, there are 24 uncertainty instances, affecting the flight.

[30] https://github.com/openai/baselines
[31] http://ardupilot.org/copter/

ArduPlane (AP)[32] is an autonomous control system for fixed-wing aircraft, and it is instrumented with two **self-healing behaviors**. One is a rule-based policy for handling network disruption between an aircraft and its ground controller. When the response time from the controller is over a threshold, the aircraft is considered as disconnected with the controller, and then the behavior will control the aircraft to fly back and land on the launching place. The other self-healing behavior is a control algorithm used to avoid collision with a nearby aerial vehicle. AP uses four sensors and one actuator, the same with ArduCopter, to locate and manipulate an aircraft. Thus, its behaviors are also affected by 24 uncertainty instances.

Table 23 Uncertainties in the Subject Systems

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

ArduRover (AR)[33] is an open-source autopilot system for ground vehicles. It has two self-healing behaviors. One is a control algorithm for avoiding collisions. The other is a rule-based policy that helps a vehicle to drive back when it is disconnected with its radio controller. Totally, ArduRover employs three sensors (one accelerometer, one GPS, and one rangefinder) and one actuator (a motor) to control a vehicle. Since a ground vehicle runs on the ground, ArduRover only monitors and controls two dimensions of the vehicle,

[32] http://ardupilot.org/plane/
[33] http://ardupilot.org/rover/

i.e., longitude and latitude. Thus, there are two instances for each type of uncertainty of the sensors and actuator. In total, ArduRover is affected by 14 uncertainty instances.

Adaptive Production Cell (APC) [26] is an autonomous manufacturing unit, which is comprised of three robots and three carts that deliver workpieces among the robots. A workpiece is to be processed in three steps, referred as three tasks for the robots. Every robot is equipped with three tools for accomplishing the three tasks. As it takes time for a robot to switch its tools, the three robots are configured to work together, and each of them only performs one of the tasks. APS is equipped with one self-healing behavior: a rule-based policy that reassigns tasks among robots to maintain the normal function of the production cell, in case one or multiple tools of a robot break. Due to inaccurate positions, delivered by the carts, and measured by the robots' sensors, APC is affected by 18 uncertainty instances: 3 carts × 3 instances of Position Deviation + 3 robots × 3 instances of Position Accuracy.

RailCab (RC) [27] is an autonomous railway system, whose function is to make rail vehicles drive in convoy to reduce energy consumptions. Driving in convoy requires the vehicles to maintain a small distance between each other, and thus it is crucial to keep a correct speed and direction of all vehicles in convoy. RC employs two speed sensors and eight eddy current sensors to measure the speed and steering direction of a vehicle. The self-healing behavior of RC is a quantitative model-based algorithm, used to detects errors caused by malfunction of the speed sensors used by a vehicle. In case the errors are detected, the behavior will identify the fault and adapt the vehicle to use GPS rather than speed sensors to measure its speed. As shown in Table 23, the movement is affected by 18 uncertainty instances, arising from the 11 sensors and one actuator.

Mobile Robot (MB) [28] [29] is an autonomous robot control system for directing a robot to play soccer. In normal operation mode, MB controls three omnidirectional wheels to move the robot. Three self-healing behaviors are implemented in MB. Two of the behaviors are control algorithms that are used to detect the incorrect movement caused by the fault that a wheel becomes stuck or a wheel rotates freely, and make the robot still follow a desired trajectory in case the fault happens. The remaining self-healing behavior is a rule-based policy used to detect and restart malfunctioning software services. As there are strong dependencies among the services, the self-healing behavior has to find a correct order to stop and start involved services. In the system, a robot is equipped with a laser

scanner to locate a soccer, a sonar range finder to measure the distance to the soccer, and a motor for movement. In total, eight uncertainty instances are impacting the behaviors of a robot, with two instances of Direction Accuracy from the scanner, two instances of Distance Accuracy from the range finder, and four uncertainty instances from the motor.

Testing these self-healing behaviors is challenging, as it is needed to decide when and which fault is to be introduced to test the self-healing behavior. As a fault may occur at any time during execution, the set of all possible test cases is huge. It could be infeasible to cover all cases. To effectively find cases in which a self-healing behavior will fail, we have proposed the executable model-based testing in our previous work [1]. In this approach, we test an SH-CPS against a test model, by executing them together, sending them the same testing stimuli, and comparing their consequent states. In this way, no test cases need to be generated before test execution. Additionally, by learning from the results of performed stimuli, reinforcement learning algorithms can be applied to learn the best policies of choosing stimulus to more effectively detect unexpected behaviors.

### 3.3.2 Test Models

By applying MoSH (a modeling framework for testing SH-CPS under uncertainty) [1], the first author of this paper built the test models[34] for the selected six SH-CPSs. For each system, we first captured its main components (e.g., sensors, actuators, and controllers) as UML classes, and then specified the components' state variables and testing interfaces as properties and operations. For each component, we further specified its expected behaviors as state machines. Based on the requirement of each system (summarized in Table 24), we defined state invariants for all the states in the state machines.

Note that the state of an SH-CPS comprises the states of its components, and the behaviors of all components form the SH-CPS behavior. With a flattening algorithm [2], the components' behaviors can be combined into a single state machine, representing the behavior of the SH-CPS. Table 25 presents descriptive statistics of the combined state machine for the six SH-CPSs.

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

Table 24 Requirements of the Subject Systems Used for Deriving State Invariants

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

Table 25 Descriptive Statistics of Behaviors of the Subject Systems

| Category | Subject System | # States | # Transitions | # Uncertainties | Average Model Execution Time (second) | Average System Execution Time (min) |
|---|---|---|---|---|---|---|
| Real-world Systems | ArduCopter (AC) | 432 | 1396 | 24 | 12 | 8 |
| | ArduPlane (AP) | 96 | 270 | 24 | 3 | 6 |
| | ArduRover (AR) | 140 | 650 | 12 | 8 | 10 |
| Systems from literature | Adaptive Production Cell (APC) | 1016 | 8512 | 18 | 15 | 3 |
| | RailCab (RC) | 2160 | 13310 | 18 | 18 | 5 |
| | Mobile Robot (MR) | 1080 | 4656 | 8 | 15 | 3 |

As explained in Section 2, the specified test models are executable, and they are executed together with the SH-CPSs for testing. The last two columns of Table 25 show the average time taken to execute a test model alone and the time taken to execute an SH-CPS (software part) with simulators of sensors, actuators, and environment, for one episode. As the software of the system and the simulation models used by the simulators are complex, executing an SH-CPS is computationally expensive. Consequently, compared with executing a test model, it is much slower to execute an SH-CPS.

## 3.4 Tasks

To assess the failure detection abilities and scalabilities of the reinforcement learning algorithms, we apply them to test the six SH-CPSs, check the effectiveness and efficiency of the algorithms, and calculate their time and space cost. An algorithm's performance depends on its capability of learning, while it also relies on the number of episodes (i.e.,

testing runs) that an algorithm can take to detect failures, as well as the range of each uncertainty.

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

On the other hand, the range of each uncertainty will affect its impact on an SH-CPS. For instance, if the range of an measurement error is extremely small, it will have little impact on the measurement. In contrast, if the measurement error is sufficiently large, it may lead to an incorrect measurement, which may increase the risk of abnormal behavior. The ranges presented in Table 23 were defined based on the product specifications of the sensors and actuators, whereas the actual ranges of measurement errors and actuation deviations could differ from the specifications. To account for the effect of the ranges, we chose to test each SH-CPS with 10 sets of ranges, which includes the set of ranges shown in Table 23 as the standard ranges, and nine sets of ranges derived by increasing or reducing the standard ranges by 10, 20, 30, 40, and 50 percent. We use 10 scales, i.e., 60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150%, to represents these 10 sets of ranges. The scale of 100% represents the standard range, 60% denotes the ranges reduced by 40%, and 150% means the ranges increased by 50%.

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

## 3.5 Hypotheses and Variables

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

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

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

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

   $$Effect(APP_i, SCALE_s, ENUM_e) = Effect(APP_j, SCALE_s, ENUM_e)$$

   Alternative hypothesis,

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

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

- $APP_i$: one of the 14 testing approaches
- *APP_SET:* the set of the 14 testing approaches, {*APPQ_A3C, APPQ_TRPO, APPQ_UPO, APPQ_PPO, APPQ_DDPG, APPS_ACKTR, APPS_DDPG, APPS_A3C, APPS_ACER, APPS_PPO, APPS_TRPO, APPS_UPO, APPQ_ACKTR, APPQ_ACER*}[35]
- *SCALE_SET*: the set of uncertainty scales, {60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150%}
- *ENUM_SET*: the set of numbers of episodes, {1000, 2000, 3000, 4000, 5000}
- *Effect* represents an effectiveness metric, which could be *SCov, TCov*, or *NDF*

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

- *SCov* is the percentage of states that are covered by a number of episodes. It is calculated by: $SCov(APP_i, SCALE_s, ENUM_e) = \frac{\left|\bigcup_{k=1}^{ENUM_e} S_{k,APP_i,SCALE_s}\right|}{|S_{all}|}$, where $S_{k,APP_i,SCALE_s}$ represents the set of states, visited by $APP_i$ in the k[th] episode, under uncertainty scale $SCALE_s$; $S_{all}$ represents the set of all states in a test model.
- *TCov* is the percentage of covered transitions. Similar with *SCov, TCov* is calculated by $TCov(APP_i, SCALE_s, ENUM_e) = \frac{\left|\bigcup_{k=1}^{ENUM_e} T_{k,APP_i,SCALE_s}\right|}{|T_{all}|}$, where $T_{k,APP_i,SCALE_s}$ is the set of transitions, visited by $APP_i$ in the k[th] episode, under uncertainty scale $SCALE_s$; $T_{all}$ represents the set of all transitions.
- *NDF* is calculated by:$NDF(APP_i, SCALE_s, ENUM_e) = \sum_{k=1}^{ENUM_e} FD_{k,APP_i,SCALE_s}$, where $FD_{i,APP_i,SCALE_s}$ denotes whether a failure is detected by $APP_i$ in the k[th] episode, under uncertainty scale $SCALE_s$. $FD_{i,APP_i,SCALE_s}$ equals 1 if a failure is detected, otherwise, 0.

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

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

   $$EFF(APP_i, SCALE_s, ENUM_e) = EFF(APP_j, SCALE_s, ENUM_e)$$

   Alternative hypothesis,

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

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

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

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

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

In summary, the empirical study involves three independent variables and six dependent variables. Table 26 summarizes their values and mapping to the goals.

Table 26 Independent and Dependent Variables

| Variable Type | Variable Name | Value | Mapping to Goals |
|---|---|---|---|
| Independent Variable | *APP* | $APP_{Q\_A3C}$, $APP_{Q\_TRPO}$, $APP_{Q\_UPO}$, $APP_{Q\_PPO}$, $APP_{Q\_DDPG}$, $APP_{S\_ACKTR}$, $APP_{S\_DDPG}$, $APP_{S\_A3C}$, $APP_{S\_ACER}$, $APP_{S\_PPO}$, $APP_{S\_TRPO}$, $APP_{S\_UPO}$, $APP_{Q\_ACKTR}$, $APP_{Q\_ACER}$ | Goal 1, Goal 2 |
| | *SCALE* | 60%, 70%, 80%, 90%, 100%, 110%, 120%, 130%, 140%, 150% | |
| | *ENUM* | 1000, 2000, 3000, 4000, 5000 | |
| Dependent Variable | *SCov* | Real Number | Goal 1 |
| | *TCov* | Real Number | |
| | *NDF* | Integer Number | |
| | *EFF* | Real Number | |
| | *TCost* | Real Number | Goal 2 |
| | *SCost* | Real Number | |

## 3.6 Statistical Tests

Table 27 summarizes the statistical tests and related variables used to evaluate the effectiveness, efficiency, time cost, and space cost of the 14 combinations of reinforcement learning algorithms. We first tested the normality of the samples of dependent variables (*SCov*, *TCov*, *NDF,* and *EFF*), using the Shapiro-Wilk test [30] with a significance level of 0.05. Test results showed that the samples are not normally distributed. Therefore, we chose to use non-parametric statistics, the Kruskal-Wallis test [31] and the Dunn's test [32] in conjunction with the Benjamini-Hochberg correction [33], to check statistical significances, and applied the Vargha and Delaney statistics [34] to measure effect sizes.

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

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

Table 27 Overview of Statistical Tests with Goals

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

# 4 Experiment Execution

We introduce the hyperparameter settings of the reinforcement learning algorithms in Section 4.1 and the experiments' execution process in Section 0.

## 4.1 Hyperparameter Tuning

Although reinforcement learning algorithms have demonstrated great learning abilities in multiple fields [10, 35, 36], the success of a particular learning algorithm depends upon the joint tuning of the model structure and optimization procedure [37]. Both of them are controlled by several hyperparameters, such as a neural network's structure, learning rate, loss function, and the number of episodes. The hyperparameters impact the whole learning process, and must be tuned to fully unlock an algorithm's potential. However, hyperparameter tuning is computationally expensive and time-consuming. In the context of software testing, testers may not have a sufficient time budget to tune an algorithm for every system under test. It will be helpful to have a hyperparameter setting that can achieve relatively good performance for a wide range of systems.

Reinforcement learning researchers have recommended several default hyperparameter settings [38]. However, these settings were tuned for playing computer games, which are different from the testing problem. Due to the high computational cost of hyperparameter tuning and limited computational resources we have, we could not afford to use all systems with all variables' settings to tune the hyperparameter. Among the six selected SH-CPSs, AP is the simplest, with the least number of states and transitions, while RC is the most complex one, and AC is a moderate one. We chose to use these three systems with diverse complexities to do the tuning, to make the selected systems more representative. For the value of the uncertainty scale and the number of episodes, we chose a moderate setting for

tuning, i.e., the uncertainty scale of 100% and the number of 3000, so as to avoid achieving a hyperparameter setting performing well only in extreme cases.

We applied the Population-Based Training (PBT) method [37] for tuning. Compared with sequential optimization or parallel grid/random search, PBT can focus on the hyperparameter space that has a higher chance of producing good results, and thus more efficiently find a better hyperparameter setting. For each reinforcement learning algorithm used in the experiment (Section 3.2), PBT was allowed to take maximally 1000 iterations to find the optimal hyperparameter setting. During the search, the three systems (AC, AP, and RC) were used to evaluate the performance of a setting, with the uncertainty scale of 100% and the number of episodes of 3000. The setting that leads to the highest fragility was regarded as the optimal solution. Table 28 presents the optimal hyperparameter setting we found for each algorithm.

Table 28 Overview of Hyperparameter Settings

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

## 4.2 Execution Process

As explained in Section 3, we applied 14 combinations of reinforcement learning algorithms to test six SH-CPSs with 10 uncertainty scales and five settings of episodes numbers. The experiment was conducted on Abel, a cluster at the University of Oslo[36]. Each testing job was run on eight nodes with 32 GB RAM. The whole empirical study took more than six months' worth of execution time.

We measured the state coverage (*SCov*), transition coverage (*TCov*), number of detected failures (*NDF*), time cost (*TCost*), and space cost (*SCost*) for each approach and each testing task. The approaches' efficiencies (*EFF*s) were further calculated, using *NDF* and *TCost*. At last, we applied statistical tests to assess the differences of the measurements among the 14 approaches.

# 5 Experiment Results

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

## 5.1 Effectiveness

For most of the testing tasks, the 14 testing approaches managed to cover all the states and transitions of the SH-CPS under test, i.e., the state coverage (*SCov*) and transition coverage (*TCov*) equal to 100%. The exceptions are the tasks for testing RC, APC, and MR. For testing APC and MR with 1000 episodes, only 74.2% and 69.4% transitions were covered, on average. When the number of episodes increased to above 2000, all of the approaches managed to cover all transitions. For testing RC, none of the 14 approaches achieved 100% transition coverage, and only a few approaches got 100% state coverage in very few cases, as RC has huge sets of states and transitions (Table 25). As an example, Fig. 14 presents the box plots of the state and transition coverages of the approaches for testing RC. The p-values of the Kruskal-Wallis test in terms of the state and transition coverages for all the testing tasks are greater than 0.1, thereby indicating no significant difference among the 14 approaches regarding the coverages.

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

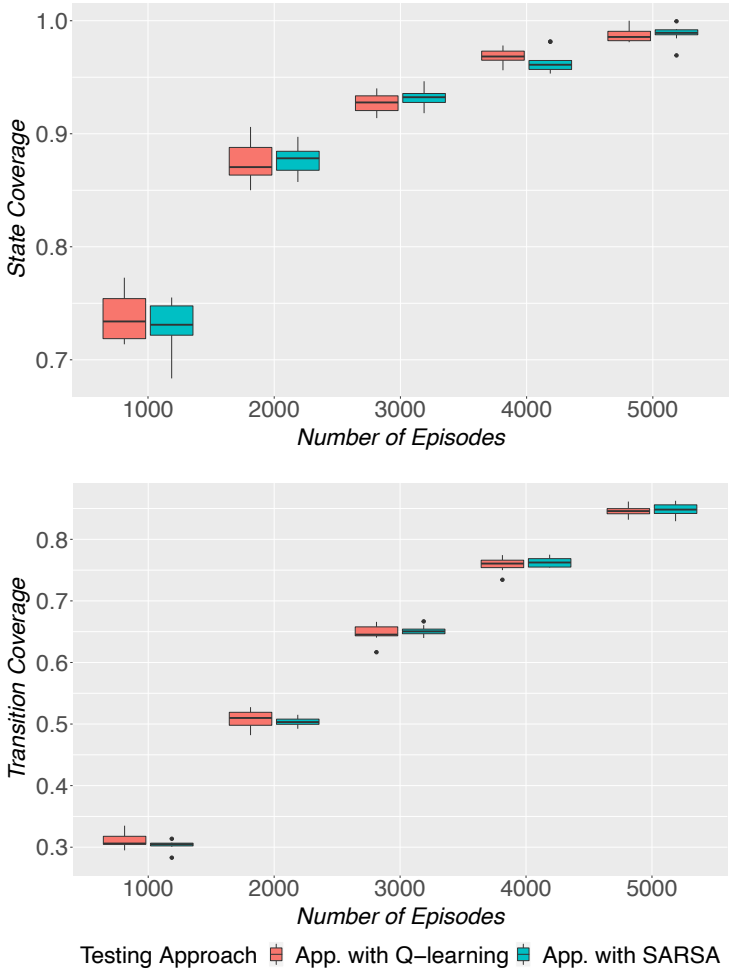


Fig. 14 State and Transition Coverages for RC

Next, we assess the actual failure detection ability of the approaches. There are 41 failures detected in the six SH-CPSs, with seven failures detected in AC, eight failures detected in AR, eight failures detected in AP, five failures detected in APC, eight failures detected in RC, and five failures detected in MR. These 41 failures correspond to 41 states in which their invariants were violated when the six systems were being tested with simulated sensors and actuators. Examples of these failures include a collision between a copter and an intruding vehicle, a crash of a plane on the ground, and a collision between a rover and an obstacle. Table 29 presents the average number of detected failures (*NDF*) for the 14 testing approaches, under 10 uncertainty scales and five numbers of episodes.

As shown in the table, $APP_{Q\_UPO}$ managed to detect the most failures. On average, it detected 3.4 failures in a testing task. $APP_{S\_UPO}$ performed slightly worse, with 3.3 failures

detected averagely. In contrast, the other 12 approaches only detected 1.7 failures, on average, and only detected three failures or less in most of the testing tasks.

Table 29 Average Numbers of Detected Failures (*NDF*) by the 14 testing Approaches

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

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

We first conducted the Kruskal-Wallis test to determine whether there are significant differences in the *NDF*s among the 14 approaches. The *p*-value of the Kruskal-Wallis test is less than $10^{-13}$, and thus significant differences do exist. Afterward, we applied the Dunn's test together with the Benjamini-Hochberg correction to check the significance of

the difference in *NDF*s between each pair of approaches. The effect size of the difference was also evaluated, using the Vargha and Delaney statistics $\hat{A}_{12}$. Since $APP_{Q\_UPO}$ detected the most failures, we focused on checking if this superiority is statistically significant.

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

In over 239 (out of 300) testing jobs, $APP_{Q\_UPO}$ significantly outperformed the other 12 testing approaches, except $APP_{S\_UPO}$. $APP_{Q\_UPO}$ and $APP_{S\_UPO}$ performed equally in 279 jobs; $APP_{S\_UPO}$ beat $APP_{Q\_UPO}$ in 2 jobs and $APP_{Q\_UPO}$ was superior in the other 19 jobs. Table 30 in Appendix A presents detailed results. When the maximum number of episodes (*ENUM*) is 1000, all testing approaches performed almost the same, while, as *ENUM* increases, $APP_{Q\_UPO}$ and $APP_{S\_UPO}$ exceeded the others. When the uncertainty scale (*SCALE*) is above 120%, $APP_{Q\_UPO}$ detected more failures than $APP_{S\_UPO}$ did in nine jobs, and they performed almost on the same level in other cases.

## 5.2 Efficiency

We evaluate the efficiency of the reinforcement learning algorithms, to find the combination of algorithms that takes the shortest time to detect failures. Fig. 15 shows the time taken by the algorithms to execute an SH-CPS from its initial state to a final state once. Particularly, the time cost includes the time taken to select operations and uncertainty values, generate test input, invoke corresponding testing interfaces, execute the system, evaluate the fragility of the consequent states, and use the fragility to update the Q function and uncertainty policy. On average, the testing approaches took less than 150 seconds to complete one episode. The differences among the average execution times of the different testing approaches are small, within 10 seconds. However, for different SH-CPSs, *SCALE*s, and *ENUM*s, the execution time varies a lot, ranging from 53 seconds to 480 seconds.

Compared with the approaches using SARSA, the approaches with Q-learning took less time to perform one episode.

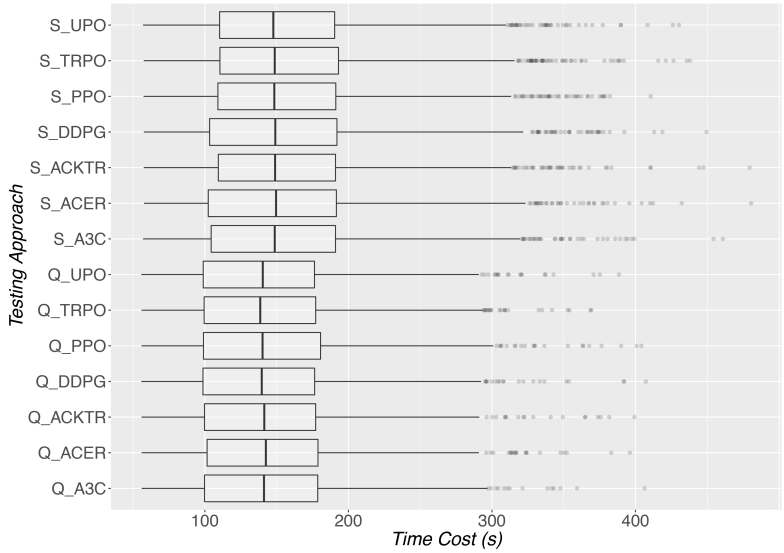


Fig. 15 Total Time Cost for One Episode

As the time taken to execute the system depends on the system's implementation, rather than the performance of the algorithms, Fig. 16 presents the time cost, excluding the time spent for executing the system. On average, the algorithms took about 12 seconds in one episode. Consistent with the trend revealed by Fig. 15, Fig. 16 also shows that SARSA related approaches took a longer time to perform an episode. In general, $APP_{Q\_A3C}$ incurred the least time cost, though the differences are within 5 seconds.

Based on the time costs and the number of detected failures (NDF), shown in Section 5.1, we calculated the efficiency measure (EFF), shown in Fig. 17. Unsurprisingly, $APP_{Q\_UPO}$ took the least amount of time to detect a failure, since all testing approaches took a similar amount of time and $APP_{Q\_UPO}$ detected the most failures within this time. Averagely, $APP_{Q\_UPO}$ took 64.5 hours to detect a failure, which is less than half of the average time taken by $APP_{Q\_A3C}$ (the least efficient approach) for failure detection. On average, $APP_{Q\_UPO}$ took 52% less time than the other approaches to detect a failure.

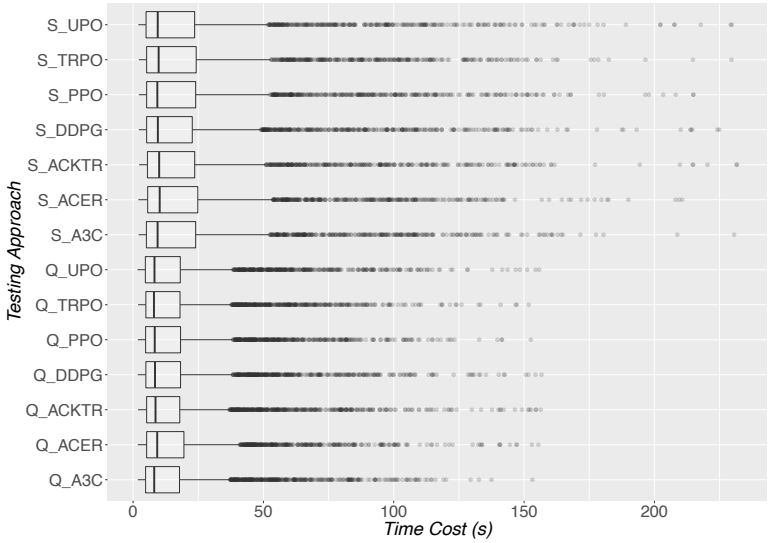


Fig. 16 Algorithm Related Time Cost for One Episode

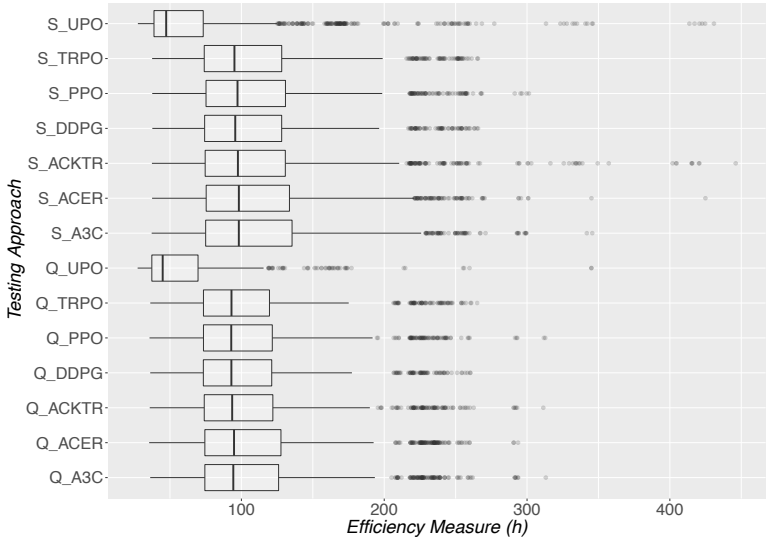


Fig. 17 Testing Approaches' Efficiencies for 300 Testing Jobs

To evaluate the significance of the differences, we conducted the Kruskal-Wallis test. The *p*-value of the test is less than $10^{-10}$, and thus there are significant differences in the *EFF*s among different testing approaches. We further applied the Dunn's test together with the Benjamini-Hochberg correction to examine if *EFF*s of $APP_{Q\_UPO}$ are significantly smaller than *EFFs* of the other approaches. The Vargha and Delaney statistics $\hat{A}_{12}$ was used to assess the effect size. Compared with $APP_{S\_UPO}$, $APP_{Q\_UPO}$ took significantly less time for failure detection in 98 jobs, more time in one job, and performed equally well in 201 jobs. For the other testing approaches, $APP_{Q\_UPO}$ was significantly more efficient in over 238 jobs. Table 31 in Appendix B presents more details.

## 5.3 Scalability

We first assess the tendencies of time and space costs of the 14 testing approaches for learning the policy of choosing operation invocations, i.e., learning how to trigger the outgoing transitions of each state defined in a test model to maximize the fragility of the SH-CPS under test. Fig. 18 shows the average time cost of each testing approach per episode, and Fig. 19 presents their average space costs. In the figures, the systems are shown in the increasing order of numbers of states and transitions, i.e., AP has the least states and transitions, and RC has the most. As shown in the figures, all costs exhibit the same tendency: the more states and transitions an SH-CPS has, the more time and space a testing approach took to learn the optimal policy of invoking operations for failure detection. For the simplest subject system, AP, which contains 96 states and 270 transitions, the 14 testing approaches took about 10 seconds to perform one episode, and used 5 GB memory space on average. In contrast, for the most complex system, RC, with 2160 states and 13310 transitions, the testing approaches' time and space costs raised to about 70 seconds and 15 GB respectively.

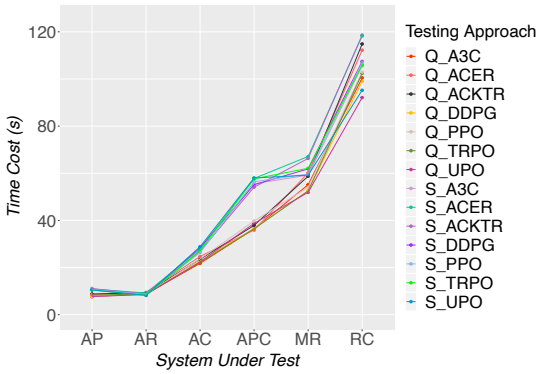


Fig. 18 Average Time Cost for Choosing and Invoking Operations

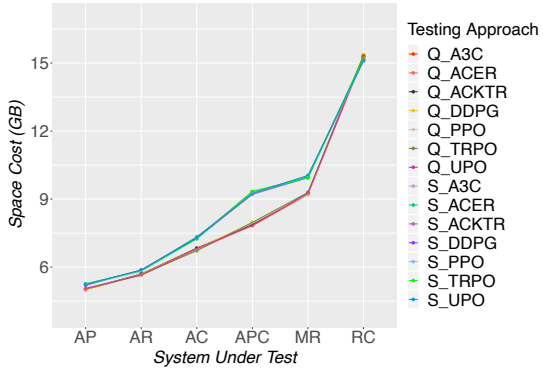


Fig. 19 Average Space Cost for Choosing and Invoking Operations

For the second task, the algorithms need to learn the policy of choosing uncertainty values that can impede the SH-CPSs and lead to failures. Fig. 20 and Fig. 21 present tendencies of their time and space costs as the number of involved uncertainties increase, where the systems are shown in the increasing order of the number of uncertainties. As one can see from the figures, the time and space costs remained at the same level for the six SH-CPSs with varying numbers of uncertainties. Because the policy based algorithms employ Artificial Neural Network (ANN) to select actions, they do not need to store the Q values of all actions for each state. Consequently, their time and space costs are fixed as long as the architectures of the ANNs are not changed. In Appendix C, Table 32 presents the detailed time and space cost of each testing approach for each SH-CPS.

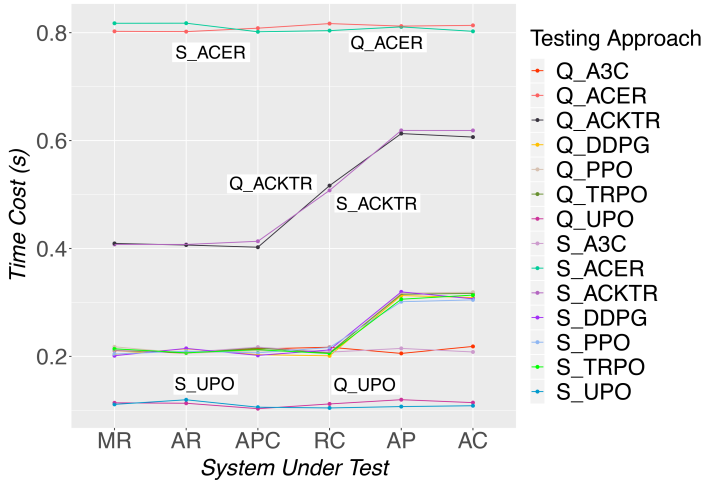


Fig. 20 Average Time Cost for Selecting and Introducing Uncertainties

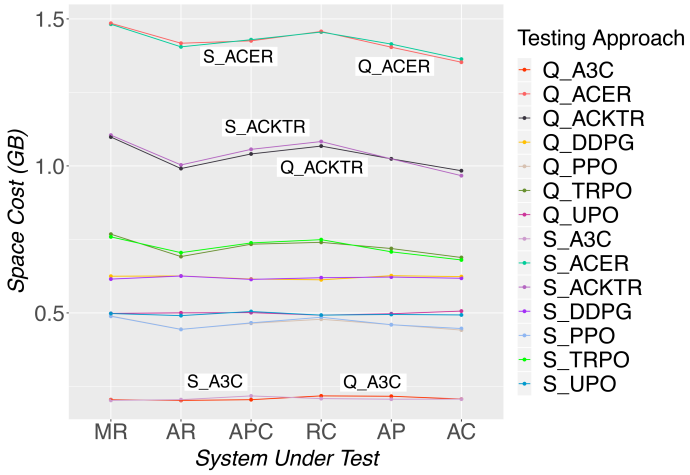


Fig. 21 Average Space Cost for Selecting and Introducing Uncertainties

# 6 Discussion

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

## 6.1 Effectiveness

Based on the results of the effectiveness, we can conclude that the combination of Q-learning and UPO is the most effective approach that detected the most failures in the six SH-CPSs, even though the 14 testing approaches achieved similar state and transition coverages. As shown in Fig. 22, when *ENUM* equals 1000, the 14 combinations of the algorithms performed at the same level. When the algorithms only had 1000 episodes to find failures, they could not collect sufficient experience from the executions, and just detected one failure on average. As the *ENUM* increased, the algorithms had more chances to explore the states of the system under test, with diverse operation invocations and uncertainty values. Consequently, the algorithms had more data to optimize their policies, and applied them to detect more failures. However, the increasing tendencies of *NDF* are different for the 14 testing approaches. The approaches with UPO tend to detect more failures than the approaches using A3C, ACER, ACKTR, DDPG, PPO, and TROP. Because the algorithms, like A3C and ACER, have to learn a value function to evaluate their policies and then update the policies based on the value function, many episodes were needed to obtain data for learning the value function, whenever the policy is updated. In contrast, UPO explores the space of policy directly using a probabilistic policy, which keeps UPO on trying different sequences of uncertainty values. Whenever it finds an uncertainty sequence that leads to a higher return, it updates its policy to increase the probability of selecting such a sequence. Therefore, UPO eliminates the cost of learning value functions, and potentially covers more promising sequences of uncertainty values for failure detection.

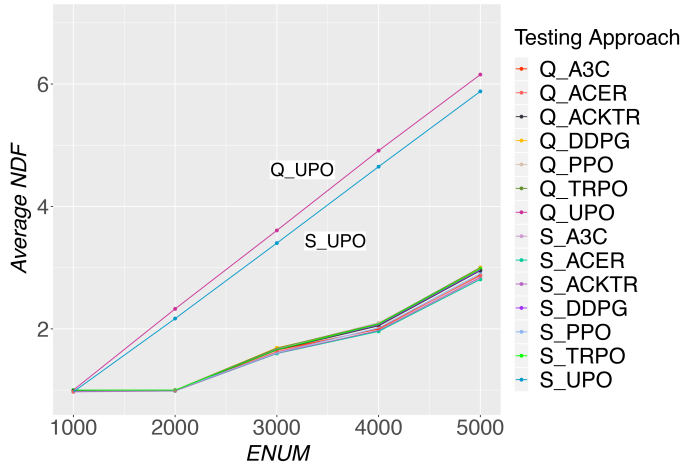


Fig. 22 Average Number of Detected Failures with Different ENUMs

Compared with *ENUM*, *SCALE* has less effect on the *NDF*. As shown in Fig. 23, the average *NDF*s of the different testing approaches decrease slightly, as *SCALE* increases. A higher *SCALE* leads to larger uncertainty values, which may cause a greater impact on system behaviors and make them more likely to fail. However, a large *SCALE* also broadens the space of uncertainty values, which makes it more difficult to find the optimal sequence of uncertainty values. Due to these two reasons, *SCALE* is only slightly affected by the *NDF*. It should also note that the ranges of uncertainty values have great impact on the performance of the testing approaches. Sufficient knowledge is required to specify the ranges correctly prior to applying the testing approaches.

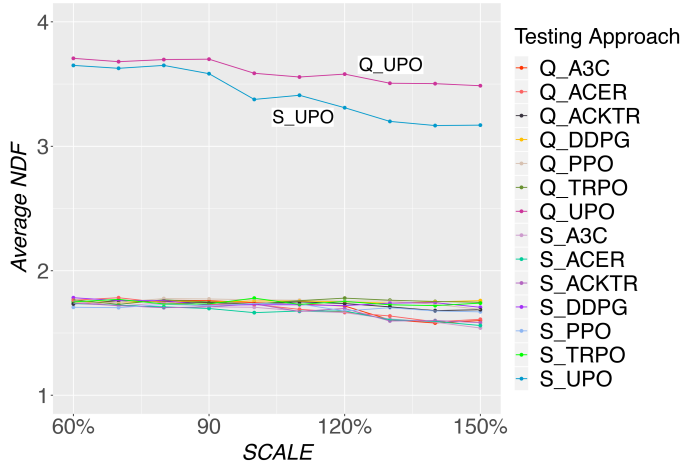


Fig. 23 Average Number of Detected Failures with Different SCALEs

## 6.2 Efficiency

Based on the results of the efficiency (Section 5.2), we know that the combination of Q-learning and UPO took the least amount of time to detect a failure. As shown in Fig. 16, on

average, the algorithms took about 12 seconds to perform one episode. The difference in the average time costs among different algorithms is within 5 seconds. For operation invocations, both SARSA and Q-learning aim to learn the optimal Q function, based on which the algorithms find the optimal policy. While the Q function is being updated, SARSA has to follow its current policy (Equation (4)), whereas Q-learning only needs to find the maximum Q value of consequent states (Equation (5)). Therefore, the computational complexity of Q-learning is less than SARSA's. For introducing uncertainty, all of the policy optimization algorithms calculate a standard gradient or natural gradient, and apply gradient descent methods to optimize their policies. Consequently, their computational complexities are similar and mainly determined by the architecture of ANNs, used as the policy and value function in these algorithms. Since, in this experiment, the ANNs used by the algorithms have almost the same number of layers and the same number of neurons in each layer (Table 28), the algorithms had very similar time costs.

## 6.3 Scalability

The results in Section 5.3 reveal that the time and space costs of learning the policy of selecting uncertainty values remain in the same order of magnitude for testing SH-CPSs with diverse complexities. In contrast, the costs of learning the policy of invoking operations are rising as the numbers of states and transitions of the system under test increase. Since the policy optimization methods were used for uncertainties and they take advantage of ANNs to approximate their policies and value functions, the computational costs of these algorithms are determined by the architecture of the ANNs and the optimizer to improve the ANNs. Alternatively, value function methods, i.e., Q-learning and SARSA used for operation invocations, have to store the Q value for each pair of state and transition explicitly. As the number of states and transitions increases, such methods will take more space and time to store and process the Q values. This could be a potential scalability issue that limits the maximum numbers of states and transitions of the SH-CPS under test. One approach to resolve this issue is to use ANNs to approximate the Q value. However, as explained in Section 3.2, for testing an SH-CPS, the candidate operation invocations are not fixed. They will change when the system switch from one state to another. As the inputs of ANNs have to be fixed, we have to train an ANN for each operation to predict its Q value. Further study is needed to determine whether multiple

ANNs could be trained together efficiently to find the optimal policy for operation invocations.

## 6.4 Alternative Approaches

The objective of testing an SH-CPS under uncertainty is to find a sequence of operation invocations and a sequence of uncertainty values that can work together to make an SH-CPS fails to behave as expected. We formulate this testing problem as an optimization problem, that is to find the optimal policies of choosing operation invocations and uncertainty values to maximize the chance of detecting failures. Through a trial and error process, reinforcement learning can learn how to select testing stimuli to reach the highest fragility and reveal a failure. Results from our previous work also demonstrated the effectiveness of RL in solving this testing problem [2].

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

In our empirical study, we applied seven policy-based reinforcement learning algorithms to find optimal uncertainty values. However, we acknowledge that evolutionary algorithms could also be applied together with reinforcement learning as demonstrated in [39, 40]. Due to limited resources, these algorithms were not included in this empirical study. It will be valuable to evaluate the performance of these algorithms in the future. Evolutionary algorithms can also be used alone to solve this testing problem. However, to test a system based on state machines, evolutionary algorithms have to find valid transition paths first, which has already been proven as a challenging task [41, 42]. A walkaround solution is to generate all valid transition paths first, according to some coverage criteria, such as all transitions, and then apply evolutionary algorithms to select a subset of paths as a test suite [43]. Nevertheless, our previous experiment results demonstrate that covering all states and

transitions is not sufficient for detecting failures in the SH-CPSs problem [2]. Consequently, we did not choose evolutionary algorithms to solve the testing problem.

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

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

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

# 7 Threats to Validity

This section analyzes the threats to validity from four aspects.

## 7.1 Construct Validity

To evaluate the failure detection ability of the 14 combinations of reinforcement learning algorithms, we took the percentages of covered states (SCov) and transitions (TCov) and the number of detected failures (NDF) as the metrics. In addition, we further defined efficiency measure (EFF), time cost (TCost), and space cost (SCost), to investigate the

efficiency and scalability of the algorithms. The metrics are comparable across the 14 combinations of algorithms, and they can directly reflect the effectiveness, efficiency, and cost of each combination.

One threat to construct validity is that the failures detected by the algorithms could be caused by potential flaws in test models rather than system defects. To mitigate the threats, first, we have defined four UML profiles to extend UML class diagrams and state machines [1]. Stereotypes defined in the profiles enable us to precisely specify expected functional behaviors, abnormal behaviors due to faults that occurred at runtime, self-healing behaviors for handling faults, and uncertainties that will affect these behaviors. Meanwhile, state invariants were used to define the valid ranges of state variables. The invariants enable us to rigorously define what behaviors are expected for a given state.

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

## 7.2 Internal Validity

As explained in Section 2.3, we chose to test SH-CPSs in a two-steps approach, as it can reduce the search space an algorithm has to explore to find the optimal solution. Nevertheless, additional experiments are still needed to verify if the two-steps approach is the best choice. Based on this two-steps approach, we evaluated the performance of 14 combinations of algorithms. The effectiveness and efficiency of a combination of reinforcement learning algorithms depend on the complexity of the system under test, the ranges of uncertainties that impact the system, the number of episodes the algorithms can take to detect failures, and hyperparameter settings of the algorithms.

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

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

## 7.3 Conclusion Validity

Due to the indeterminate policy used by the reinforcement learning algorithms to explore different operation invocations or uncertainty values, the number of detected failures and space/time costs are affected by randomness, threatening the conclusion validity. To reduce the threat, we repeated each testing job 10 times and applied statistical tests to evaluate the significance of the experiment result. We conducted the Kruskal-Wallis test [31] and Dunn's test [32] in conjunction with the Benjamini-Hochberg correction [33] to check statistical significance, and Vargha and Delaney statistics [34] to measure effect size. Finally, we acknowledge that more repetitions are needed to increase the trust on the results further.

## 7.4 External Validity

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

# 8 Related Work

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

## 8.1 Testing with Reinforcement Learning

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

In a pioneering work, Veanes et al. devised an ad hoc reinforcement learning algorithm for online testing [44]. With the aim of covering more system behaviors, the algorithm keeps track of the number of times a transition has been triggered, and chooses a transition that has been triggered with the least of times. In their experiments, the ad hoc algorithm was compared with a random testing algorithm, and the proposed algorithms managed to cover more states than the random one, with much less time. However, the ad hoc algorithm does not consider the long-term reward, that is, the coverage of future transitions. Thus, the policy learned by this algorithm may be suboptimal.

In another work, Groce et al. proposed a light-weight automated testing framework for container-like classes [45]. In their framework, SARSA (a value function learning algorithm, see Section 2.4.1) was used to learn the policy of generating test cases, i.e., sequences of method calls on container objects. In the evaluation, the SARSA based approach was compared with random testing and a modeling checking approach for 15 container classes. Their evaluation results show that the new approach performed better in 7 out of the 15 classes. As no other reinforcement learning algorithms were evaluated in the experiment, it is unknown whether other algorithms will perform better.

Mariani et al. and Reichstaller et al. applied Q-learning for GUI [46] and interoperability testing [47]. In the first work, Q-learning was used to select the testing action that maximizes the changes of displayed GUI widgets, to cover functions of a system under test. In their empirical evaluation, the Q-learning based approach was compared with GUITAR, an open-source GUI testing tool, for four GUI applications. For all of these applications, the Q-learning based approach achieved a higher code coverage, and detected more faults than GUITAR. In the second work, Q-learning was applied to find implementation faults

that can lead to the most critical failures, so that the riskiest implementation faults can be tested. The proposed testing approach was only evaluated by applying it for one case study, without comparing it with other methods.

Spieker et al. proposed a value function learning based reinforcement learning algorithm, similar to Q-learning, for test case prioritization [48]. In the evaluation, the reinforcement learning-based approach was compared with a random and two static test case prioritization approaches. Evaluation results demonstrated that the reinforcement learning-based approach could effectively learn to prioritize test cases that have a high chance to detect faults, with performance comparable with the two static methods, within 60 iterations.

More recently, Reichstaller and Knapp proposed a model-based reinforcement learning algorithm for testing self-adaptive systems [49]. Different from the reinforcement learning algorithms evaluated in this empirical study, the model-based algorithm tries to learn a Markov Decision Process (MDP) model of the system under test. An MDP model is defined by 1) a set of states of the system, 2) a set actions that can be performed on the system, 3) the transition probability that the system switches from one state to another when an action is conducted, and 4) the reward of performing an action under a state. When the MDP model is learned, it can be used to find the optimal policy of taking actions to maximize cumulative rewards. In the evaluation, the model-based algorithm was compared with Q-learning and a random method for testing a smart vacuum system. Testing results reveal that the model-based algorithm performed the best, and both model-based algorithm and Q-learning outperformed the random method. However, sufficient domain knowledge is needed to obtain the MDP model, and the current algorithm only supports learning the transition probability for a low-dimensional state space. These limitations restrict the applicability of the model-based reinforcement learning algorithm, and it needs further research to enhance the generalizability and learning capability.

In summary, existing works mainly evaluated the performance of value function learning based reinforcement learning algorithms for test case generation, prioritization, and risk-based testing. Besides, the hyperparameter settings used in these works, and how the hyperparameter settings were selected, were rarely mentioned in these papers. To find the optimal reinforcement learning algorithms for testing SH-CPSs under uncertainty, we conducted this empirical study and evaluated the performance of 14 combinations of

reinforcement learning algorithms. By tuning these algorithms and applying them to test six SH-CPSs, we found the optimal reinforcement learning algorithms that detected the most failures in these systems, and with the least time cost.

## 8.2 Testing under Uncertainty

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

For uncertainty mitigation, Zhang et al. and Ji et al. both proposed to use Model-Based Testing (MBT) to discover unknown system behaviors due to indeterminate environmental conditions [50] or uncertain networks [51]. In another work [52], Camilli et al. also applied MBT to collect actual system responses at runtime, and then the responses are fed to a Bayesian inference process that updates beliefs on uncertain parameters of system behaviors, modeled as a Markov Decision Process (MDP). Based on the result of the Bayesian inference process, values of the uncertain parameters are calibrated, and the calibrated MBP model can be used to support future development. From another perspective, Walkinshaw and Fraser proposed an uncertainty-driven Learning Based Testing (LBT) approach for unit testing [53]. In this approach, Walkinshaw and Fraser apply genetic programming to learn multiple inference models of the program under test, based on previous testing results. Then, an active learning technique (Query By Committee) is used to select a test input, for which the inference models are the most uncertain about the outputs. The test input is then used to test the program, and the actual output of the program is used to further update the inference models, which are used to choose the next test input. Unlike these works that aim to discover unknown system behaviors or mitigate uncertainty, our work aims to find failures in SH-CPSs under a set of already identified uncertainties (measurement errors and actuation deviations), with the range of each uncertainty given.

To enable testing under uncertainty, Menghi et al. proposed an approach to generate test oracles for testing Simulink models with uncertain parameter values and white noises. In this approach, functional requirements are specified as Restricted Signals First Order Logic (RFOL) formulas and the formulas are transformed to Simulink blocks to calculate a quantitative measure, representing the degree of satisfaction of the requirements.

Alternatively, in our work, we use a test model to capture the requirements of the system under test, and the constraints defined in the test model serve as test oracles. Simulation is also a common approach used to test systems under uncertainty. Ramirez et al. proposed to use simulators of sensors to test a goal model used by an adaptive system, with the measurement of sensors affected by noises and failures [54]. Similarly, Minnerup and Knoll proposed to use simulators of actuators to test automated vehicles against a set of actuator inaccuracies [55]. In these two works, the options of uncertainty are limited to either a few types, durations and severities of noises [54] or several samples of actuator inaccuracies, sampled from their ranges [55]. On the contrary, our work aims to find a value within a valid range for each uncertainty and for each measurement or actuation, the uncertainty may take effect. Since the solution space of our testing problem is huge, we proposed to use reinforcement learning to effectively find the sequence of uncertainty values that can reveal a failure.

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

# 9 Conclusion

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

increasing the numbers of states and transitions of the system under test will incur extra space and time costs for SARSA and Q-learning, which were used for operation invocations. Whereas increasing the number of uncertainties has little effect on the costs of the other algorithms, which were used for introducing uncertainties.

# Acknowledgement

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

# References

[1] T. Ma, S. Ali, and T. Yue, "Modeling foundations for executable model-based testing of self-healing cyber-physical systems," *Software & Systems Modeling,* vol. 18, no. 5, pp. 2843-2873, 2019.

[2] T. Ma, S. Ali, T. Yue, and M. Elaasar, "Testing self-healing cyber-physical systems under uncertainty: a fragility-oriented approach," Software Quality Journal, vol. 27, no. 2, pp. 615-649, 2019.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," arXiv preprint arXiv:1708.05866, 2017.

[4] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction (no. 1). MIT press Cambridge, 1998.

[5] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in International conference on machine learning, 2016, 2016, pp. 1928-1937.

[6] Z. Wang et al., "Sample efficient actor-critic with experience replay," arXiv preprint arXiv:1611.01224, 2016.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.

[8] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in Pro*ceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015: PMLR, 2015, pp. 1889-1897.

[9] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in Advances in neural information processing systems, 2017, 2017, pp. 5279-5288.

[10] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," arXiv preprint arXiv:1509.02971, 2015.

[11] Semantics Of A Foundational Subset For Executable UML Models V1.2.1, OMG, 2016.

[12] Precise Semantics Of UML State Machines (PSSM), OMG, 2017.

[13] Object constraint language V2.0, OMG, 2006.

[14] S. Ali, M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Generating Test Data From OCL Constraints With Search Techniques," IEEE Transactions on Software Engineering, vol. 39, no. 10, pp. 1376-1402, 2013.

[15] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in International Conference on Machine Learning, 2016, 2016, pp. 1329-1338.

[16] B. Kiumarsi, K. G. Vamvoudakis, H. Modares, and F. L. Lewis, "Optimal and autonomous control using reinforcement learning: A survey," IEEE transactions on neural networks and learning systems, vol. 29, no. 6, pp. 2042-2062, 2017.

[17] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, "Convergence results for single-step on-policy reinforcement-learning algorithms," Machine learning, vol. 38, no. 3, pp. 287-308, 2000.

[18] J. Martens and R. Grosse, "Optimizing neural networks with kronecker-factored approximate curvature," in International conference on machine learning, 2015, 2015, pp. 2408-2417.

[19] D. Pollard, "Asymptopia: an exposition of statistical asymptotic theory," ed, 2000.

[20] R. Pascanu and Y. Bengio, "Revisiting natural gradient for deep networks," arXiv preprint arXiv:1301.3584, 2013.

[21] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare, "Safe and efficient off-policy reinforcement learning," in Advances in Neural Information Processing Systems, 2016, 2016, pp. 1054-1062.

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in software engineering. Springer Science & Business Media, 2012.

[23] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in Guide to advanced empirical software engineering: Springer, 2008, pp. 201-228.

[24] B. A. Kitchenham et al., "Preliminary guidelines for empirical research in software engineering," IEEE Transactions on software engineering, vol. 28, no. 8, pp. 721-734, 2002.

[25] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri, "A review of process fault detection and diagnosis: Part I: Quantitative model-based methods," Computers & chemical engineering, vol. 27, no. 3, pp. 293-311, 2003.

[26] M. Güdemann, F. Ortmeier, and W. Reif, "Safety and dependability analysis of self-adaptive systems," in *Leveraging Applications of Formal Methods, Verification and Validation. ISoLA 2006. Second International Symposium on*, 2006: IEEE, 2006, pp. 177-184.

[27] C. Priesterjahn, D. Steenken, and M. Tichy, "Timed hazard analysis of self-healing systems," in Assurances for Self-Adaptive Systems: Springer, 2013, pp. 112-151.

[28] G. Steinbauer, M. Mörth, and F. Wotawa, "Real-time diagnosis and repair of faults of robot control software," in Robot Soccer World Cup, 2005: Springer, 2005, pp. 13-23.

[29] M. Brandstotter, M. W. Hofbaur, G. Steinbauer, and F. Wotawa, "Model-based fault diagnosis and reconfiguration of robot drives," in IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007: IEEE, 2007, pp. 1203-1209.

[30] P. Royston, "Remark AS R94: A remark on algorithm AS 181: The W-test for normality," Journal of the Royal Statistical Society. Series C (Applied Statistics), vol. 44, no. 4, pp. 547-551, 1995.

[31] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," Journal of the American statistical Association, vol. 47, no. 260, pp. 583-621, 1952.

[32] O. J. Dunn, "Multiple comparisons using rank sums," Technometrics, vol. 6, no. 3, pp. 241-252, 1964.

[33] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," Journal of the royal statistical society. Series B (Methodological), pp. 289-300, 1995.

[34] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," Journal of Educational and Behavioral Statistics, vol. 25, no. 2, pp. 101-132, 2000.

[35] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, p. 529, 2015.

[36] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," The International Journal of Robotics Research, vol. 32, no. 11, pp. 1238-1274, 2013.

[37] M. Jaderberg et al., "Population based training of neural networks," arXiv preprint arXiv:1711.09846, 2017.

[38] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in Thirty-Second AAAI Conference on Artificial Intelligence, 2018, 2018.

[39] S. Khadka and K. Tumer, "Evolution-guided policy gradient in reinforcement learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 1188-1200.

[40] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," Journal of Machine Learning Research, vol. 7, no. May, pp. 877-917, 2006.

[41] K. A. Derderian, "Automated test sequence generation for finite state machines using genetic algorithms," Brunel University, School of Information Systems, Computing and Mathematics, 2006.

[42] P. K. Lehre and X. Yao, "Runtime analysis of the (1+ 1) EA on computing unique input output sequences," Information Sciences, vol. 259, pp. 510-531, 2014.

[43] R. Lefticaru and F. Ipate, "Automatic state-based test generation using genetic algorithms," in Ninth international symposium on symbolic and numeric algorithms for scientific computing (synasc 2007), 2007: IEEE, pp. 188-195.

[44] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," Formal Approaches to Software Testing and Runtime Verification, pp. 240-253, 2006.

[45] A. Groce et al., "Lightweight automated testing with adaptation-based programming," in IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE), 2012: IEEE, 2012, pp. 161-170.

[46] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in Software Testing, Verification and Validation (ICST), IEEE Fifth International Conference on, 2012: IEEE, 2012, pp. 81-90.

[47] A. Reichstaller, B. Eberhardinger, A. Knapp, W. Reif, and M. Gehlen, "Risk-Based Interoperability Testing Using Reinforcement Learning," in IFIP International Conference on Testing Softw*are and Systems*, 2016: Springer, 2016, pp. 52-69.

[48] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017: ACM, 2017, pp. 12-22.

[49] A. Reichstaller and A. Knapp, "Risk-based Testing of Self-Adaptive Systems using Run-Time Predictions," in IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2018: IEEE, 2018, pp. 80-89.

[50] M. Zhang, S. Ali, and T. Yue, "Uncertainty-wise Test Case Generation and Minimization for Cyber-Physical Systems: A Multi-Objective Search-based Approach," Journal of Systems and Software, 2019.

[51] R. Ji et al., "Uncovering unknown system behaviors in uncertain networks with model and search-based testing," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018: IEEE, pp. 204-214.

[52] M. Camilli, A. Gargantini, and P. Scandurra, "Model-based hypothesis testing of uncertain software systems," Software Testing, Verification and Reliability, vol. 30, no. 2, p. e1730, 2020.

[53] N. Walkinshaw and G. Fraser, "Uncertainty-driven black-box test data generation," in 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017: IEEE, pp. 253-263.

[54] A. J. Ramirez, A. C. Jensen, B. H. Cheng, and D. B. Knoester, "Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems," in Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, 2011: IEEE Computer Society, pp. 568-571.

[55] P. Minnerup and A. Knoll, "Testing Automated Vehicles against Actuator Inaccuracies in a Large State Space," IFAC-PapersOnLine, vol. 49, no. 15, pp. 38-43, 2016.

# Appendix A Evaluation Results for Effectiveness

Table 30 presents the statistical test results for the *Number of Detected Faults* (*NDF*). As $APP_{Q\_UPO}$ detected the most faults, we focus on comparing $APP_{Q\_UPO}$ with the other approaches. For each of the other 13 testing approaches, denoted as $APP_c$, we checked the *p*-value of the Dunn's test, corresponding to the pair of $APP_{Q\_UPO}$ and $APP_c$. If the *p*-value is over 0.05, the two testing approaches are considered to perform equally well, denoted as "=" in Table 30. Otherwise, we further computed the Vargha and Delaney statistics $\hat{A}_{12}$, using the *NDF*s of the two approaches. If $\hat{A}_{12}$ is above 0.5 for the pair of $APP_{Q\_UPO}$ and $APP_c$, it means $APP_{Q\_UPO}$ has a higher chance to detect more faults, and thus $APP_{Q\_UPO}$ is considered to be superior to $APP_c$, signified as ">". Otherwise, $APP_{Q\_UPO}$ is considered to be worse, signified as "<".

Table 30 Statistical Test Results for Effectiveness

| SCALE | ENUM | $APP_{Q\_UPO}$ **vs.** | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $APP_{Q\_A3C}$ | | | $APP_{Q\_ACER}$ | | | $APP_{Q\_PPO}$ | | | $APP_{Q\_TRPO}$ | | | $APP_{Q\_ACKTR}$ | | | $APP_{Q\_DDPG}$ | | |
| | | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = |
| 60% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 70% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 80% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 90% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 100% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 110% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 120% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 130% | 1000 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 140% | 1000 | 1 | 0 | 5 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 150% | 1000 | 1 | 0 | 5 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 5 | 0 | 1 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| Sum | | 243 | 0 | 57 | 242 | 0 | 58 | 240 | 0 | 60 | 240 | 0 | 60 | 240 | 0 | 60 | 239 | 0 | 61 |

Table 10 Statistical Test Results for Effectiveness (continued)

| SCALE | ENUM | $APP_{Q\_UPO}$ vs. | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $APP_{S\_A3C}$ | | | $APP_{S\_ACER}$ | | | $APP_{S\_PPO}$ | | | $APP_{S\_TRPO}$ | | | $APP_{S\_ACKTR}$ | | | $APP_{S\_DDPG}$ | | | $APP_{S\_UPO}$ | | |
| | | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = |
| 60% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 1 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 70% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 80% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 90% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 100% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 1 | 0 | 5 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 110% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 120% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| 130% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 1 | 0 | 5 |
| | 2000 | 5 | 0 | 1 | 5 | 0 | 1 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 1 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| 140% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 4 | 0 | 2 | 4 | 0 | 2 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 6 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| 150% | 1000 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 5 | 0 | 1 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| SUM | | 238 | 0 | 62 | 236 | 0 | 64 | 240 | 0 | 60 | 240 | 0 | 60 | 241 | 0 | 59 | 240 | 0 | 60 | 19 | 2 | 279 |

# Appendix B Evaluation Results for Efficiency

Table 31 summarizes the evaluation results for efficiency. We focus on comparing $APP_{Q\_UPO}$ with the other approaches. If an approach $APP_c$ took significantly more (less) time than $APP_{Q\_UPO}$ to detect a fault, $APP_c$ is inferior (superior) to $APP_{Q\_UPO}$ in terms of efficiency, denoted as ">" ("<") in Table 31.

Table 31 Statistical Results for Efficiency

| SCALE | ENUM | $APP_{Q\_UPO}$ **vs.** | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $APP_{Q\_A3C}$ | | | $APP_{Q\_ACER}$ | | | $APP_{Q\_PPO}$ | | | $APP_{Q\_TRPO}$ | | | $APP_{Q\_ACKTR}$ | | | $APP_{Q\_DDPG}$ | | |
| | | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = |
| 60% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 70% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 90% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 5 | 0 | 1 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 100% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 110% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 120% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 130% | 1000 | 1 | 0 | 5 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 5 | 0 | 1 | 5 | 0 | 1 | 5 | 1 | 0 | 5 | 0 | 1 | 5 | 0 | 1 | 5 | 0 | 1 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 140% | 1000 | 0 | 0 | 6 | 1 | 0 | 5 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 5 | 0 | 1 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| 150% | 1000 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 5 | 0 | 1 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 |
| Sum | | 240 | 0 | 60 | 239 | 0 | 61 | 239 | 61 | 0 | 239 | 0 | 61 | 238 | 0 | 62 | 238 | 0 | 62 |

Table 11 Statistical Results for Efficiency (continued)

| SCALE | ENUM | $APP_{Q\_UPO}$ **vs.** | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $APP_{S\_A3C}$ | | | $APP_{S\_ACER}$ | | | $APP_{S\_PPO}$ | | | $APP_{S\_TRPO}$ | | | $APP_{S\_ACKTR}$ | | | $APP_{S\_DDPG}$ | | | $APP_{S\_UPO}$ | | |
| | | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = | > | < | = |
| 60% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 1 | 0 | 5 | 2 | 0 | 4 | 2 | 0 | 4 | 1 | 0 | 5 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| 70% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 |

| | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| 80% | 1000 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| 90% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| 100% | 1000 | 1 | 0 | 5 | 1 | 0 | 5 | 2 | 0 | 4 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| 110% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 4 | 0 | 2 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| 120% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 1 | 0 | 5 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 1 | 0 | 5 |
| 130% | 1000 | 3 | 0 | 3 | 2 | 0 | 4 | 2 | 0 | 4 | 3 | 0 | 3 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 1 | 2 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| 140% | 1000 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 | 2 | 0 | 4 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 5 | 0 | 1 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| 150% | 1000 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 1 | 0 | 5 | 2 | 0 | 4 | 1 | 0 | 5 | 3 | 0 | 3 |
| | 2000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 3000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 2 | 0 | 4 |
| | 4000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| | 5000 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 0 | 3 | 0 | 3 |
| SUM | | 258 | 0 | 42 | 257 | 0 | 43 | 257 | 0 | 43 | 257 | 0 | 43 | 257 | 0 | 43 | 259 | 0 | 41 | 98 | 1 | 201 |

# Appendix C Time and Space Costs of Reinforcement Learning Algorithms

Table 32 presents quantiles of time and space costs of each testing approach for the six SH-CPSs.

Table 32 Time and Space Costs of Each Testing Approach for Six SUTs

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | RC | 2160 | 2432 | 18 | 56.6 | 70.5 | 85.7 | 14.0 | 15.2 | 16.5 |
| | MR | 1080 | 1656 | 8 | 39.4 | 50.9 | 64.1 | 9.3 | 10.0 | 10.8 |
| S_A3C | AC | 432 | 432 | 24 | 19.5 | 28.2 | 39.2 | 6.6 | 7.3 | 7.9 |
| | AR | 140 | 360 | 12 | 6.3 | 8.3 | 10.8 | 5.3 | 5.8 | 6.4 |
| | AP | 96 | 270 | 24 | 8.0 | 10.4 | 13.9 | 4.8 | 5.2 | 5.9 |
| | APC | 1000 | 1008 | 18 | 30.7 | 52.3 | 88.6 | 8.0 | 9.2 | 10.6 |
| | RC | 2160 | 2432 | 18 | 57.4 | 69.9 | 85.5 | 13.9 | 15.1 | 16.5 |
| | MR | 1080 | 1656 | 8 | 39.1 | 51.1 | 65.0 | 9.3 | 9.9 | 10.9 |
| S_ACER | AC | 432 | 432 | 24 | 19.8 | 30.1 | 40.7 | 6.6 | 7.3 | 8.0 |
| | AR | 140 | 360 | 12 | 7.1 | 9.3 | 11.9 | 5.3 | 5.8 | 6.4 |
| | AP | 96 | 270 | 24 | 8.7 | 11.6 | 14.9 | 4.8 | 5.3 | 6.0 |
| | APC | 1000 | 1008 | 18 | 34.6 | 51.9 | 91.6 | 8.0 | 9.2 | 10.4 |
| | RC | 2160 | 2432 | 18 | 64.9 | 79.0 | 96.9 | 14.0 | 15.2 | 16.4 |
| | MR | 1080 | 1656 | 8 | 43.5 | 55.3 | 68.5 | 9.3 | 9.9 | 10.8 |
| S_PPO | AC | 432 | 432 | 24 | 18.0 | 26.9 | 38.3 | 6.6 | 7.3 | 7.9 |
| | AR | 140 | 360 | 12 | 6.3 | 8.6 | 11.1 | 5.3 | 5.8 | 6.4 |
| | AP | 96 | 270 | 24 | 7.8 | 10.4 | 13.6 | 4.8 | 5.2 | 6.0 |
| | APC | 1000 | 1008 | 18 | 34.5 | 57.7 | 91.8 | 8.0 | 9.3 | 10.4 |
| | RC | 2160 | 2432 | 18 | 58.4 | 72.3 | 88.4 | 13.9 | 15.1 | 16.4 |
| | MR | 1080 | 1656 | 8 | 39.3 | 50.7 | 64.4 | 9.3 | 10.0 | 10.8 |
| S_TRPO | AC | 432 | 432 | 24 | 19.1 | 27.3 | 38.8 | 6.7 | 7.3 | 8.0 |
| | AR | 140 | 360 | 12 | 6.3 | 8.5 | 10.9 | 5.3 | 5.8 | 6.4 |
| | AP | 96 | 270 | 24 | 7.8 | 10.9 | 14.1 | 4.8 | 5.2 | 6.0 |
| | APC | 1000 | 1008 | 18 | 31.0 | 54.8 | 95.9 | 8.0 | 9.3 | 10.5 |
| | RC | 2160 | 2432 | 18 | 57.6 | 69.0 | 89.0 | 14.0 | 15.3 | 16.4 |
| | MR | 1080 | 1656 | 8 | 40.2 | 51.0 | 65.4 | 9.3 | 9.9 | 10.8 |
| Q_ACKTR | AC | 432 | 432 | 24 | 16.2 | 22.9 | 35.0 | 5.8 | 6.8 | 7.6 |
| | AR | 140 | 360 | 12 | 7.0 | 9.1 | 11.6 | 5.3 | 5.7 | 6.1 |
| | AP | 96 | 270 | 24 | 6.6 | 8.4 | 10.8 | 4.6 | 5.0 | 5.9 |
| | APC | 1000 | 1008 | 18 | 23.8 | 39.6 | 67.7 | 6.6 | 7.9 | 9.3 |
| | RC | 2160 | 2432 | 18 | 59.0 | 72.3 | 94.4 | 13.1 | 15.3 | 17.0 |
| | MR | 1080 | 1656 | 8 | 36.5 | 45.1 | 63.5 | 8.2 | 9.3 | 9.9 |
| S_UPO | AC | 432 | 432 | 24 | 18.0 | 26.8 | 39.2 | 6.7 | 7.3 | 7.9 |
| | AR | 140 | 360 | 12 | 6.2 | 8.4 | 10.7 | 5.3 | 5.9 | 6.4 |
| | AP | 96 | 270 | 24 | 8.1 | 10.8 | 13.7 | 4.8 | 5.2 | 6.0 |
| | APC | 1000 | 1008 | 18 | 35.5 | 56.7 | 95.6 | 7.9 | 9.3 | 10.4 |
| | RC | 2160 | 2432 | 18 | 52.8 | 66.7 | 81.3 | 14.0 | 15.1 | 16.5 |
| | MR | 1080 | 1656 | 8 | 36.6 | 47.4 | 61.7 | 9.4 | 10.0 | 10.8 |
| Q_ACER | AC | 432 | 432 | 24 | 16.1 | 25.0 | 39.1 | 5.8 | 6.8 | 7.7 |
| | AR | 140 | 360 | 12 | 7.3 | 9.8 | 12.2 | 5.3 | 5.7 | 6.1 |
| | AP | 96 | 270 | 24 | 6.5 | 8.7 | 11.2 | 4.6 | 5.0 | 5.9 |
| | APC | 1000 | 1008 | 18 | 22.1 | 37.3 | 59.7 | 6.6 | 7.8 | 9.2 |
| | RC | 2160 | 2432 | 18 | 58.6 | 72.4 | 96.0 | 13.3 | 15.2 | 17.3 |
| | MR | 1080 | 1656 | 8 | 38.8 | 50.1 | 67.2 | 8.2 | 9.2 | 9.9 |

* #Unc: Number of uncertainty instances