Chen JC, Qin Y, Wang HY *et al.* Simulation might change your results: A comparison of context-aware system input validation in simulated and physical environments. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(1): 83–105 Jan. 2022. DOI 10.1007/s11390-021-1669-1

Simulation Might Change Your Results: A Comparison of Context-Aware System Input Validation in Simulated and Physical Environments

Jin-Chi Chen (陈金池), Yi Qin* (秦　逸), *Member*, *CCF*, *ACM*, Hui-Yan Wang (王慧妍), *Member*, *CCF*, and Chang Xu* (许　畅), *Senior Member*, *CCF*, *IEEE*, *Member*, *ACM*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

E-mail: chenjc@smail.nju.edu.cn; {yiqincs, why, changxu}@nju.edu.cn

Received June 1, 2021; accepted December 23, 2021.

Abstract Context-aware systems (a.k.a. CASs) integrate cyber and physical space to provide adaptive functionalities in response to changes in context. Building context-aware systems is challenging due to the uncertain running environment. Therefore, many input validation approaches have been proposed to protect context-aware systems from uncertainty and keep them executing safely. However, in contrast to context-aware systems' prevailing in physical environments, most of those academic solutions (83%) are purely evaluated in simulated environments. In this article, we study whether this evaluation setting could lead to biased conclusions. We build a testing platform, RM-Testing, based on DJI RoboMaster robot car, to conduct the physical-environment based experiments. We select three up-to-date input validation approaches, and compare their performance in the simulated environment and in the physical environment. The experimental results show that all three approaches' performance in simulated environments (improving task success rate by 82% compared with the system without the support of input validation) does differ from their performance in a physical environment (improving the task success rate by 50%). We also recognize three factors (scenario setting, physical platform and environmental model) that affect the performance of input validation approaches, based on an execution model of the context-aware system.

Keywords context-aware system, input validation, self-driving car, testing infrastructure

1 Introduction

The vision of Internetware calls a shift of software paradigm from executing in a static and closed environment to executing in a dynamic and open environment [1]. The developing of context-aware systems (a.k.a. CASs) echoes that call by integrating cyber and physical space, and providing adaptive functionalities in response to changes in context [2, 3]. Among various variants of "context" and "context-awareness", we adopt a software engineering based perspective of context [2, 4]. Specifically, context is defined as the abstract representation of relational expression over sensed context variables. Thus, context-aware systems are the systems that continually sense environmental changes, make decisions based on their preprogrammed logic, and then take physical actions to adapt to the sensed changes.

Due to their dynamic running environments, context-aware systems are error-prone [2, 5]. Environmental uncertainty is one of the main obstacles to a reliable context-aware system. Unlike traditional programs that take accurate and deterministic inputs, context-aware systems often suffer from uncertain input that

Regular Paper

Special Section on Software Systems 2021—Theme: Internetware and Beyond

A preliminary version of the paper was published in the Proceedings of Internetware 2020.

This work was partially supported by the National Natural Science Foundation of China under Grant No. 61932021, the Leading-Edge Technology Program of Jiangsu Natural Science Foundation of China under Grant No. BK20202001, the National Natural Science Foundation of China under Grant No. 61902173, and the Natural Science Foundation of Jiangsu Province of China under Grant No. BK20190299.

*Corresponding Author (Yi Qin mainly contributed to the empirical study on input validation's effectiveness in two different environments. Chang Xu mainly contributed to the comparison between the physical and simulated environments.)

cannot accurately describe the system's running environment. Such uncertain inputs are unpredictable while developing a context-aware system, potentially leading to abnormality or failure if not processed appropriately. In fact, uncertain inputs contribute to two of the most famous accidents of context-aware systems: the crashing of the Tesla self-driving car, in which the system failed to recognize a truck due to road reflections①, and the falling of Boeing 737-MAX airplane, in which the system received an incorrect angle of attack values②.

To address the uncertain input issue, many input validation approaches have been proposed to filter uncertainty from the inputs of a context-aware system [6–8]. These approaches could help developers to improve the quality of the inputs of context-aware systems from different perspectives, including detecting and repairing inconsistent context, recognizing the context from abnormal running scenarios, and validating the unfitted operational field of deep learning inputs. However, in contrast to context-aware systems' success in the physical world, most of those academic solutions are evaluated in simulated environments only. Based on a primitive review of software engineering related literature [5–16], we found that only 17% of these studies evaluated their approaches in physical environments [5, 7], and the others are only evaluated in a simulated environment, i.e., either in a simulator [6, 9–11], or with pre-collected execution traces of context-aware systems [8, 12–16]. The above result echoes a recent survey on context-aware applications, in which the authors [17] claimed that "(for real-world tests) testers may have to bear the high cost (e.g., time and money)". As such, one could ask two questions that Q1: whether the input validation approaches' performance varies in a physical environment and in a simulated environment, and Q2: whether and how the differences between a simulated environment and a physical environment affect the performance of those approaches. The answers to these two questions could help researchers to better validate the effectiveness of their proposed approaches, and demonstrate the usefulness of the approaches.

In this article, we design and conduct an empirical study to answer the two questions above. To the best of our knowledge, we are the first to investigate the evaluation of a CAS-related technique in the simulated and the physical environment. The conclusion of our study is summarized as follows: "The performance of input validation approaches for context-aware systems in a physical environment does differ from that in a simulated environment."

Technically, we design and build a testing platform, RM-Testing, to connect the simulation-based evaluation with a physical-world based one. We select the aforementioned autopilot program of the self-driving car as our target context-aware service, refit a DJI (DJ-Innovations) RoboMaster S1 robot car with additional range sensors to enrich its sensibility, and build a controller module to control the robot car with subject autopilot programs. We also implement and adapt three input validation approaches, namely ECC [6], CoMID [7], and DISSECTOR [8], to validate the environmental information pushed to the subject autopilot program. Based on the testing platform, we evaluate three input validation approaches in both the simulated and the physical environment.

We also analyze the factors that could impact the system's interaction with its running environment detailedly based on a conceptual model describing a context-aware system's execution in the environment. We recognize three major factors, namely scenario setting, environmental model, and physical platform. Therefore, we build different experimental scenarios of the urban road network and use different types of RoboMasters as the target system's platforms.

The experimental results show that these approaches improve the task success rate by 82% compared with the system without the support of input validation in a simulated environment, and improve that by 50% in a physical environment. The result of the study shows that the three recognized factors have different impacts on the selected input validation approaches' performance in different environments, among which physical platform makes the most significant influence.

In summary, this article makes the following contributions.

- We build a testing platform, RM-Testing, for evaluating input validation approaches with a context-aware system (DJI RoboMaster S1 robot car) in a physical environment.
- We analyze the factors that distinguish a simulated environment from a physical environment in detail, in terms of their interactions with a context-aware system.

① https://www.bbc.com/news/technology-48308852, Sept. 2021.

② http://knkt.dephub.go.id/knkt/ntsc_aviation/baru/2018%20-%20035%20-%20PK-LQP%20Final%20Report.pdf, Sept. 2021.

• We conduct experiments in both the simulated environment and the physical environment to measure the differences between the input validation approaches' performance in a physical environment and that in a simulated environment. We also investigate and validate the effect of each recognized factor that may lead to the differences.

Note. This article significantly extends [18]. Specifically, our major extensions include: an in-depth analysis of the difference of the interaction between context-aware systems and their running environment (in Section 4), and new experiments that investigate the impact of the recognized differences between a simulated environment and a physical environment (in Section 5). The new experiments not only echo the conclusion in the previous version [18] (i.e., the performance of the selected approaches in a physical environment is different from that in a simulated environment), but also further validate three factors (i.e., scenario setting, environmental model, and physical platform) that lead to the difference.

The remainder of this article is organized as follows. Section 2 introduces DJI RoboMaster S1 and three input validation approaches associated with this work. Section 3 gives an overview of our testing platform for the context-aware system, and how we adapt the aforementioned approaches in our platform. Section 4 introduces a conceptual model of program-environment interaction, and analyzes the environmental factors that affect such interaction. Section 5 presents an intensive evaluation in investigating the difference between the simulated and the physical environment. Section 6 discusses related work, and finally Section 7 concludes this paper.

2 Preliminaries

In this section, we first introduce the background of the autopilot program and DJI RoboMaster S1 robot car. Then we briefly describe three input validation approaches for context-aware systems.

2.1 Autopilot Program and DJI RoboMaster S1

An autopilot system typically consists of two parts: the perception system and the decision-making system [19–21]. The perception system is responsible for sensing the environment using equipped sensors, while the decision-making system analyzes the current state based on the sensed environmental information and determines the driving routes [20].

As a context-aware system, the autopilot program of a self-driving car controls the car's behavior to adapt to the sensed environmental information. Basically, the execution of an autopilot program consists of three steps [20]: 1) sensing the car's surrounding environment and receiving the environmental information; 2) making decisions on the car's future route using the pre-defined driving strategy; 3) controlling the car's direction and speed to follow the determined driving route.

RoboMaster S1 is an educational robot car designed by DJI. It is equipped with four omnidirectional wheels, which enable the robot car to move towards any direction and spin around within a small area. The robot is also equipped with a Wi-Fi module and a first-person-view (shorted as "FPV") camera, which enables one to connect the robot car to a computer and control its movement from the camera in real time. However, as an educational robot, RoboMaster S1 has very limited sensors (only four contact sensors to sense the car's collision with obstacles) and restricted programming supports (a scratch-program based UI with limited APIs). As a result, we have to refit both its hardware and software, in order to build our testing platforms for context-aware systems.

2.2 Input Validation for Context-Aware Systems

Context-aware systems leverage environmental information to provide autonomous and adaptive services. However, the uncertainty of environmental information introduces several challenges towards the failure-free context-aware systems as follows.

Inconsistent Context. The noise of sensing data weakens the system's ability to understand the environment [22]. Due to the limitation of physical measurement, the error is inevitable during the system's sensing phases, which could further affect the decision-making and action-performing phases, and finally lead the system into failure [6, 22]. Noisy data may make contexts conflict with each other, which is referred to as the context inconsistency problem [6].

Uncertain Scenarios. Context-aware systems often execute in a dynamic and open environment. Uncertainty is unavoidable in such environments that system developers have to make simplification of the complicated environment. As a result, context-aware systems facing uncertain scenarios could have their pre-defined

logic failing in the physical environment [23]. This problem is referred to as an abnormal state [7].

Unfitted Operational Field. To improve productivity and cope with infinite kinds of environmental dynamics, context-aware systems developers often hold certain assumptions on typical scenarios [9, 24, 25]. Such certain assumptions describe the operational field of a system. However, the environment might deviate from the operational field during their actual running and make the system unreliable. Deep learning (a.k.a. DL) approaches are widely used in many context-aware systems [20, 26]. Its application in context-aware systems further increases the risk of the unfitted operational field, and has been reported by recent literature [8, 27]. If a DL model's running environment and training environment are different, the inputs from the running environment may be out of the scope of the system's handling capability, resulting in a reduction in the quality of the system's executions [8]. This problem is referred to as unfitted DL model input [27, 28].

Many research efforts have been made to address the above challenges. In this work, we focus on three branches of input validation techniques, namely constraints checking, invariant checking, and DL model input pruning. From each of these branches, we select one approach to study their performance in both the simulated and the physical environment, with respect to improving the execution safety of context-aware systems. The purpose of input validation or pruning is to alleviate the impact of the deviation between the sensed readings and the actual values. Such deviation is produced during the interaction between a context-aware system and its running environment, not a system defect. As a result, an input validation approach can be treated as an enhancement of the context-aware system's original application logic, which improves the availability and reliability of the system [6–8]. In the following part of this subsection, we briefly introduce the selected input validation techniques.

Constraints checking addresses the context inconsistency problem by validating contexts [6, 29]. It helps prevent such inconsistency from being received by the system. Constraints describe the restrictions on the relationship between multiple pieces of contexts [30].

In this work, we study the performance of the Entire Constraints Checking (ECC) [6] approach in both the simulated and the physical environment. We use a constraint language based on first-order logic [6] to specify consistency constraints. The syntax of the constraint language is as follows:

$$f ::= \forall\gamma \in S(f)|\exists\gamma \in S(f)|(f) \wedge (f)|(f) \vee (f)| \\ (f) \rightarrow (f)|\neg(f)|bfunc(\gamma, \ldots, \gamma),$$

where $bfunc$ represents a user-defined function. The parameters of $bfunc$ are context instances and the return value is a boolean variable.

Invariant checking addresses the abnormal state problem by automatically generated invariants [7, 31]. Uncertain scenarios introduce an unpredictable situation beyond the capability of pre-defined logics of context-aware systems. Many systems use assertions to check whether the systems' surrounding environments have entered abnormal states. However, manually specified assertions can generally only detect obvious failure but not potential abnormal state.

One promising way to detect potential abnormal state is to conduct automatically generated invariant checking in the runtime. Before the system is put into use, we can automatically generate invariants that the program should satisfy when it runs normally. The violation of any invariant means that the system may enter a failure state soon, and thus the system can take corresponding repair measures to prevent failure.

Invariant detectors like Daikon [32] achieve great results in traditional software testing. Some researchers also proposed invariant generation templates for robotic systems [33]. In this work, we study the performance of Context-Aware Multi-Invariant Detection (CoMID) approach [7].

DL model input pruning addresses the unfitted DL model input problem by validating and pruning inputs to the DL model [8]. In recent years, deep learning approaches have been widely used in many context-aware systems to assist their recognition of the physical environment [20, 26, 34]. Most of these approaches use supervised learning. They need to be trained using the data collected from certain scenarios. As a result, the system's input from the running environment may be out of the model's capability [8].

In response to this problem, some DL model input validation approaches for deep learning have been proposed. In this work, we study the performance of DISSECTOR [8] approach in both the simulated and the physical environments. The main idea of this approach is to distinguish and prune the inputs that exceed the model's handling capability to prevent the data from being used in the decision-making phase. Since the remaining inputs are within the system's capability, the

system can be more reliable. More specifically, DISSECTOR tracks how the model interprets its input and generates a PVscore to denote the input's validity. The value range of the PVscore is [0,1]. An input is more likely to be within the capability of the model (i.e., being valid) if its PVsocre is closer to 1.

2.3 Testing in Simulator and Physical World

Despite the seemingly straightforward answers, a rigorous investigation requires well technical design and considerable implementation efforts. On one hand, one has to build a testing infrastructure that runs not only in a simulator but also in a physical environment. The testing infrastructure should connect the concerned program with a physical platform capable of sensing the surrounding environment and taking physical actions. As such, the testing infrastructure requires refitting and modifying the physical platforms to enrich their sensibility and controllability. To support different input validation approaches that may leverage various program analysis techniques, the infrastructure should also support recording the execution information of the context-aware system.

On the other hand, the investigation requires an in-depth comparison of a simulated environment and a physical environment, in terms of their interactions with context-aware systems. Considering various factors impacting the program-environment interaction, a broad comparison might not necessarily reveal the environments' differences. For example, a complex environmental scenario might cause a dramatic increase of the context-aware system's abnormal rate in both the two environments [35], which could further narrow the measured gap between the two environments.

Besides the efforts to design and implement, another challenge for answering the two questions is to deploy the concerned input validation approaches on a physical context-aware system. Different from a cyber laboratory environment that is closed and static, the physical environment is open and dynamic since computation is moved "off the desktop" [4, 7, 9, 36], which could make the direct application of the approaches less effective [4]. For example, DISSECTOR [8], an input validation approach for deep learning applications, drops inputs under the validity threshold directly to achieve better accuracy performance [8]. However, when it comes to a robot car recognizing the road sign by a deep learning approach, the same strategy can lead to failure if no image can reach validity. This problem prevents us from using the original tools proposed by the researchers directly.

3 Testing Platform for Context-Aware Systems

We build a testing platform, RM-Testing, on DJI RoboMaster S1. The architecture of the platform is shown in Fig.1. Basically, the platform connects a DJI RoboMaster S1 robot car and a subject autopilot program under test. The platform mainly consists of four modules, namely sense, control, input validation, and info. The first two modules enable the subject program to sense the robot car's surrounding environment and control the robot car to move and rotate, respectively. The input validation module implements three selected input validation approaches, and provides high-quality environmental information to the subject program. The info module collects execution information from all the other modules for analyzing the program's execution states.

In the following parts of this section, we will first describe RM-Testing's sense, control, and info modules in detail. Then, we will show our implementation of the concerned input validation approaches (i.e., the input validation module) on RM-Testing.

3.1 Implementation of RM-Testing

Module sense enables the subject autopilot program to acquire environmental information. As discussed in Section 2 that RoboMaster S1 is only equipped with four contact sensors, we refitted the robot car to enrich its sensibility. We install range sensors in the forward, rear, left, and right of the robot car to obtain the horizontal distance between the robot car and other objects. Since RoboMaster S1 does not open its underlying development board interface and wireless network module, we also install an Arduino UNO 3 (and its power supply device) and an ESP8266 WIFI module for data transmission. The Arduino UNO 3 is responsible for sending signals to each sensor to trigger distance measurement, waiting for response signal, and calculating the current reading based on the time difference between two signals. The ESP8266 WIFI module sends the data to a computer in LAN.

Besides the sensors' data, we also modify the robot car's FPV controller to enable the autopilot program to acquire the FPV camera's images. The autopilot program can use encapsulated approaches to control the camera to monitor its surrounding environment better.

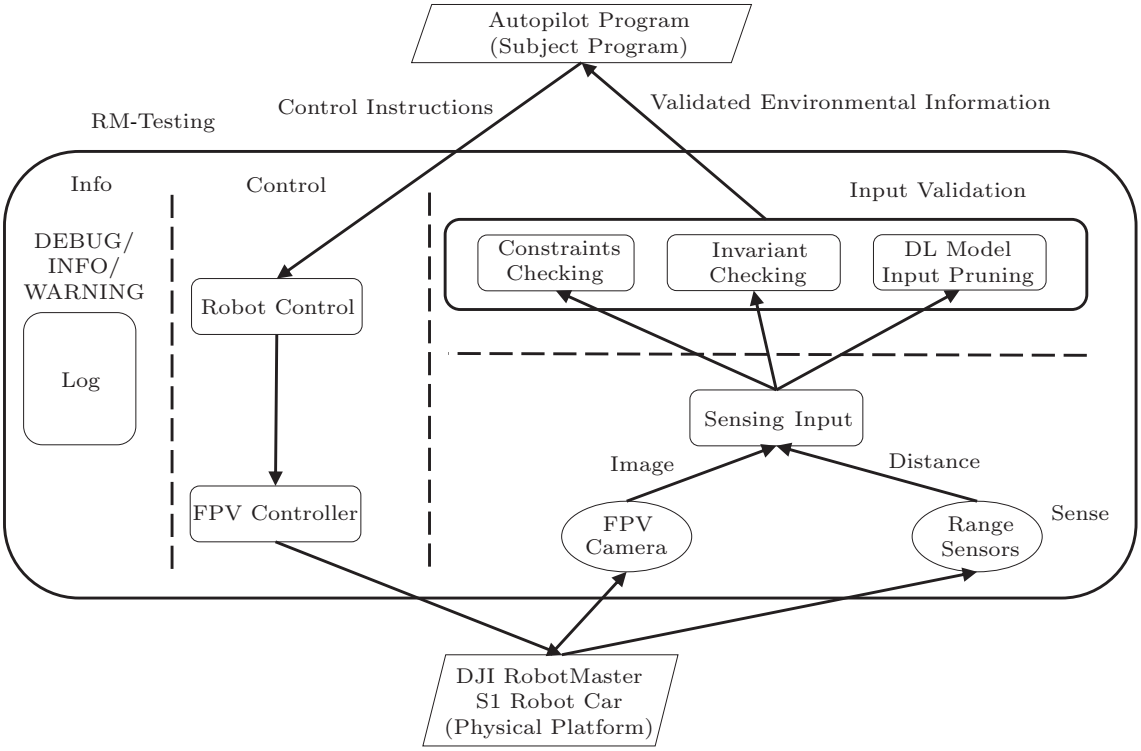


Fig.1. Architecture of RM-Testing platform [18].

The raw environmental information, such as sensor readings and images, is managed by a sensing input module. This module collects all raw environmental information from the hardware, such as sensors and the camera, and feeds it to other modules that require the data. The sensing input module connects providers (i.e., sensors and a camera) and the consumers (i.e., the input validation module and the autopilot program) of environmental information in a pub/sub manner. More specifically, the hardware that provides environmental data and the modules that require environmental data first register their names and related data types in the sensing input module. Then the sensing input module would collect the raw data from the providers and push the collected data to the consumers. This module uses FIFO queues to store the collected environmental information, considering the different producing/consuming speeds of the data.

3.2 Control in RM-Testing

Module control enables the autopilot program to control the robot car to move with enriched APIs, compared with the original APIs provided by the RoboMaster S1 IDE. We implement a robot control module based on the FPV controller of the robot car. With the original FPV controller, users can use the keyboard to move and rotate the robot car. More specifically, pressing "W", "S", "A", "D", "left", and "right" keys would control the robot car to move forward, back, left, and right, and rotate counterclockwise and clockwise, respectively. To enable the automatic control, we use the pywin32③ library to simulate keyboard actions of the FPV mode, and encapsulate those simulated keyboard actions as methods for the autopilot program to invoke as shown in Fig.2.

(a) (b) (c)

Fig.2. Refitting and control of the car [18]. (a) Before refitting. (b) After refitting. (c) FPV control.

Module info is a log-recording module that facilitates program debugging, error location, and data analysis. It is implemented based on the Python logging library. The execution record of any module in the

③https://github.com/mhammond/pywin32, Sept. 2021.

platform will be outputted to the log files. We use three logging levels, including "DEBUG", "INFO", and "WARNING". The "DEBUG" level concerns the updated information of corresponding variables in RM-Testing. The "INFO" level describes the pre-defined events produced by the RM-Testing platform. The "WARNING" level presents the checking results produced by the implemented approaches in the input validation module. We implement different levels of logs for the universality of this platform.

3.3 Input Validation in RM-Testing

Constraints Checking. In this module, we implement and modify the ECC approach for constraints checking with the RoboMaster S1 robot car. We design specific constraints that are effective for the robot car scenario based on previous studies [6, 29]. An effective constraint needs to deal with errors that frequently occur in the scenario to improve the quality of the validated context efficiently. In addition, the constraints should avoid missing detections (i.e., false-negative instances) and false alarms (i.e., false-positive instances). Thus, to design effective constraints, we first check the execution traces of the robot car and analyze the main reasons for its failure. Based on the observation, we design three types of constraint templates, which mainly concern the rapid changes of the range sensors' readings. After determining the constraint templates, we try with different parameter settings and select the best one according to their performance.

We use the following constraints in the system.

1) The change of the forward range sensor's reading should not exceed m meters within 1 second:

$$\forall\gamma_1 \in S_{\text{forward}}(\forall\gamma_2 \in S_{\text{forward}}(\\ |\gamma_1.t - \gamma_2.t| < 1 \rightarrow |\gamma_1.dist - \gamma_2.dist| < m)).$$

2) The change of the left range sensor's reading should not exceed n meters within 1 second:

$$\forall\gamma_1 \in S_{\text{left}}(\forall\gamma_2 \in S_{\text{left}}(\\ |\gamma_1.t - \gamma_2.t| < 1 \rightarrow |\gamma_1.dist - \gamma_2.dist| < n)).$$

3) The change of the right range sensor's reading should not exceed n meters within 1 second:

$$\forall\gamma_1 \in S_{\text{right}}(\forall\gamma_2 \in S_{\text{right}}(\\ |\gamma_1.t - \gamma_2.t| < 1 \rightarrow |\gamma_1.dist - \gamma_2.dist| < n)).$$

In these constraints, constants m and n can be specified manually according to the observed execution traces. These constraints are mainly used to alleviate problems like random errors and sudden changes due to transient sensor failure. When a constraint is found to be violated, we repair the consistency error with the drop-latest strategy. Once the context instance is validated, the constraint checking module stores it in a buffer for the autopilot program to use.

We use a stack for reserving contexts that may be used later for constraint checking. Let us take constraint 1 as an example. It concerns two pieces of context produced by the forward range sensor. The subject context-aware system, which uses the latest context/sensor readings, could also fetch the values of the historical context from the stack.

Invariant Checking. In this module, we implement and modify the CoMID approach for invariant checking with the RoboMaster S1 robot car. The main challenge to applying the CoMID approach is to design an effective invariant template. The original CoMID approach uses Daikon invariant inference engine④ to derive invariants. However, in our RM-Testing platform, Daikon is less effective for two reasons. On the one hand, Daikon requires instrumenting the subject program to record the execution traces, while we cannot assume the availability of the subject program's source code in our RM-Testing platform. On the other hand, Daikon's invariant templates are designed for the internal variables of a program, while our environmental invariants mainly focus on the external variables of the environment.

As a result, we have to design our invariant templates for the RoboMaster S1 robot car. Similar to the design of the constraint template in the constraint checking module, we observe execution traces of the robot car to determine the templates of the environmental invariants. We also optimize the settings of the invariants' parameters to make the generated invariants neither too general nor too specific.

We use the following invariants in the system. These invariants mainly focus on preventing the car from crashing into any obstacle.

1) When the car is about to turn, it is not too close to the obstacles on the left or right:

$$d_{\text{left}} \geqslant a \wedge d_{\text{right}} \geqslant a,$$

where variable d_{left} (d_{right}) is the reading of the left

④http://plse.cs.washington.edu/daikon, Jan. 2021.

(right) range sensor, and constant a is derived from pre-collected traces.

2) When the car is moving forward, the readings of the range sensors on the left or right do not change too rapidly:

$$\left|\frac{d_{\text{left1}} - d_{\text{left2}}}{\Delta time}\right| \leqslant b \wedge \left|\frac{d_{\text{right1}} - d_{\text{right2}}}{\Delta time}\right| \leqslant b,$$

where variable d_{left1} (d_{right1}) is the reading of the left (right) range sensor at $t1$, variable d_{left2} (d_{right2}) is the reading of the left (right) range sensor at $t2$, $\Delta time$ is $|t1 - t2|$, and constant b is derived from pre-collected traces.

We also design the remedy actions for the autopilot program to invoke in order to correct the execution of the robot car when any invariant is violated. When invariant 1 is violated, the remedy action will control the robot car to move left or right to keep away from obstacles. When invariant 2 is violated, the remedy action will control the robot car to rotate counterclockwise or clockwise to prevent deviation.

DL Model Input Pruning. In this module, we implement and modify the DISSECTOR [8] approach to validate the input of deep learning models. More specifically, we use TensorFlow to implement the DISSECTOR approach, and train a base image recognition model and five sub-models for each scenario in each environment. Each sub-model is associated with the specific layer of the base model. The system calculates a PVsocre for each input image based on the base model and five sub-models.

With the original DISSECTOR approach [8], there is a fixed threshold to determine whether the input image is valid and drops the invalid ones. In some situations, such a fixed threshold could lead the autopilot program to abort all received images during a period of time. If this happens during the robot car's passing at an intersection, the autopilot program will fail to respond to a road sign. To prevent this situation, we design a slide-window based approach for using the DISSECTOR approach. Instead of setting a fixed threshold on the image's PVscore, we perform DISSECTOR on five images at one time and choose the image with the highest PVscore to be recognized by the autopilot program.

4 Comparison Between the Simulated and Physical Environments

We will present the Program-Environment Interaction Model (PEIM) in Subsection 4.1, demonstrate a concrete context-aware system based on the PEIM model in Subsection 4.2, and propose three factors in the context of the previous illustrative program in Subsection 4.3. These factors may affect the behavior of the target context-aware system and the concerned input validation approaches' performance accordingly. They will be used as experimental factors (i.e., unfixed independent variables) in our later evaluation. This section provides a theoretical basis for the evaluation design in Section 5.

4.1 Program-Environment Interaction Model

To better analyze the environment's impact on a context-aware system, we use a PEIM model to describe the interaction between such a program and its running environment. In contrast to traditional program models that only concern programs themselves, the proposed model concerns not only the program, but also its environment under interaction [7]. Note that our PEIM only captures the iterative nature of a context-aware system in interactions with its environment.

Given a program P, we define its PEIM using a tuple, (P, E, $Inter$, C). We use P to represent the program, and E represents the environment where the program executes. Conceptually, we consider environment E as a black-box program whose behavior can be observed by monitoring its global variables, although one may not know how E works. We assume that one can observe P's behavior in E (i.e., P's output) and P's obtained sensory data from E (i.e., P's input). We use C to represent P's initial configuration (e.g., default startup parameter values) and E's initial configuration (e.g., initial environmental layout or scenario/map settings).

We use $Inter$ to represent the interface that connects P's input/output with E's output/input. Specifically, $Inter$ can be described as a function that maps environment E's output O_E to program P's input I_P. If one does not consider uncertainty, I_P would trivially equal O_E. However, in practice, $I_P \neq O_E$ is due to uncertainty. Their differences are caused by inaccurate environmental sensing (e.g., a sensed value deviates from its supposed value) or flawed physical actions (e.g., an action is taken without exactly achieving its supposed effect) [37]. It may be impossible for the developers to derive a complete specification that could tell the exact values of the differences caused by the uncertainty in any iteration. In this case, the developers could assume or derive a partial specification of the uncertainty (e.g.,

the distribution followed by the variable, such as normal distribution or Poisson distribution). Therefore, we assume that $Inter$ leverages a partial specification of uncertainty, which contains information on ranges and distributions of uncertainty on the conversion between I_P and O_E values.

As a whole, a PEIM, (P, E, $Inter$, C) works in an iterative way, as illustrated in Fig.3. It starts with program P and environment E initialized by configuration C (step 1). Then both P and E begin their independent executions. At the program side, P gets its input I_P from the environment's current output O_E, executes based on I_P, updates its global variables G_P, and finally returns output O_P (step 2). At the environment side, E takes its input I_E from the program's current output O_P, "executes" by applying I_E's effect to update its global variables G_E, and finally returns output O_E (step 3). Once O_P or O_E is produced, E or P receives it, converts it to I_E or I_P, and puts the result in a buffer for later use. When P or E finishes its iteration, it obtains its next input I_P or I_E from the corresponding buffer using some policy, e.g., FIFO or priority-first (an input for indicating that an emergency situation can be processed first). We conceptually represent the impact of uncertainty on the conversion between P and E by $I_P = Inter(O_E)$ (step 4). Steps 2–4 form an iterative reaction loop.

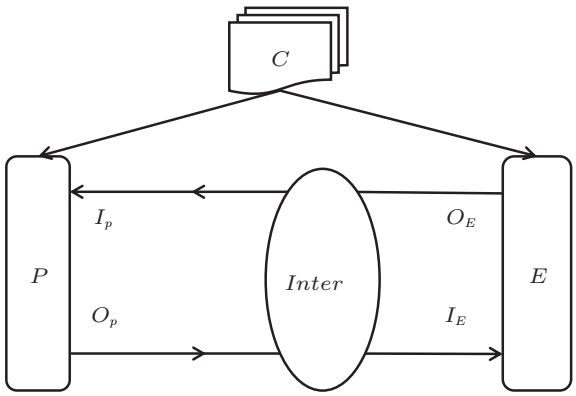


Fig.3. PEIM's iterative reaction loop.

4.2 Autopilot Program

Considering our target context-aware service of autonomous driving, a DJI RoboMaster is controlled by an autopilot program P. The basic logic of program P is shown in Algorithm 1. Constraint checking is used in another thread which updates the sensor reading variables; therefore it is not shown in Algorithm 1.

The environment E, according to our PEIM, describes the program's running environment. E takes the RoboMaster's actions (e.g., action type $type_A$ and parameter $para_A$ for action A) as input, updates its environmental states (e.g., the position and posture of the robot), and produces P's sensory data as output. For interface $Inter$, the mapping between I_P/I_E and O_E/O_P values is determined by both inaccurate sensing, which maps a given environment's output parameter o_E to an error range $[o_E - lower_E, o_E + upper_E]$ for P to sense, and flawed physical action, which also maps a given action parameter $para_A$ to an error range $[para_A - lower_A, para_A + upper_A]$. The configuration C specifies the initial states of P (e.g., each approach is enabled or disabled) and E (e.g., the initial position of the robot, and the layout of the obstacles).

Algorithm 1. Autopilot Program P

```
# Enable or disable each approach
constraint_checking_enabled ← true
invariant_checking_enabled ← true
dl_pruning_enabled ← true
# Keep going straight and turning
while true do
    while not reach_intersection() do
        move_a_unit();
        if invariant_checking_enabled then
            r ← invariant_checking()
            if not r then
                invariant_repair()
            end
        end
    end
    # If parameter is true, use DISSECTOR
    d = get_direction(dl_pruning_enabled)
    turn(d)
end
```

4.3 Differences Between Simulated and Physical Environment

Our PEIM model describes the elements that interact between a context-aware system P and its running environment E. As such, we recognize three factors that differ between a simulated environment and a physical environment, in terms of their interaction with a context-aware system. These factors, namely scenario setting, environmental model, and physical platform, correspond to C, E, and $Inter$ in the PEIM model, respectively.

The scenario setting describes the impact of different configurations on program-environment interaction. Although configuration C does not directly involve PEIM's iterative execution, it implicitly affects a context-aware system's behavior by changing P's and E's initial states. Since P always receives its inputs

from the surrounding environment E, different initial states of P and E will result in different input sequences that P receives. Thus, a well-chosen C favored by P might cause E to produce high-quality inputs that are unlikely to trigger P's abnormal behavior. Such inputs will not trigger the activation of the concerned input validation approaches, and make one unable to measure the performance of those approaches.

In the case of our target autopilot program, we consider the difficulty for P to control the robot car to avoid crashing into the obstacles. In a complicated scenario full of obstacles and intersections, the RoboMaster's crashing may not be caused by the poor quality of P's received inputs, but may be caused by the complex internal logic to control the robot car. As a result, we cannot directly represent the concerned input validation approaches' performance by the observed P's abnormal rate. The same is true in simple scenarios that are free from obstacles and intersections, in which a robot car is unlikely to crash into any obstacles even when P's received inputs are of inferior quality.

The environmental model describes the internal logic of how E produces its output O_E based on a given I_E. In fact, E's internal logic involves physical laws that determine E's reaction to P's output and E's updates of states. For a physical environment, one could regard it as a black box program that follows natural physical laws. For a simulated environment, it depends on the laws that are pre-defined by the simulator designers. As a result, facing the same i_E values, the gap between a physical environment's physical laws and a simulated environment's physical laws could result in different o_E, and thus affects P's sensed i_P values. During P's continuous interaction with E, such a difference would accumulate and cause P's different behaviors.

With our example of RoboMaster, the environmental model explains how P's control instructions are performed by the robot car, and how P's sensed input values should be updated according to its movement. Since we build our simulated environment within the Unity system, our simulated environment actually uses its internal physical engine to map the robot car's actions to its future sensor inputs. If the physical engine does not reflect the natural physical laws accurately, then P's execution trace in the simulated environment might deviate from its execution trace in the physical environment. Such deviation makes the direct comparison between P's abnormal rates in the simulated environment and the physical environment less effective in demonstrating the difference between the two environments.

The physical platform involves the interface that connects P with E. Similar to the environmental model, the physical platform also reflects the physical laws. While the environmental model concerns the mapping between i_E and o_E values, the physical platform concerns the inaccurate sensing and flawed physical actions that affect the mapping between i_P/i_E and o_E/o_P values. Existing work on context-aware systems often assumes that such mappings caused by uncertainty are subject to normal distributions. As a result, a simulated environment usually injects uncertainty by randomizing an accurate value by normal distribution [38]. When the actual uncertainty in the physical environment no longer subjects to the normal distribution, or subjects to a normal distribution of different mean and variance values, both P and E will receive different uncertain input values in the two environments.

To study the impact of the physical platform on the concerned approaches' performance, we consider different types of RoboMaster cars. Besides the RoboMaster S1 described in Section 3, we also implement RM-Testing with RoboMaster EP, a revised version of robot cars designed by DJI. The major differences between RoboMaser S1 and EP include the sensors and the control APIs. S1 uses ultrasonic sensors and a keyboard-based API to control the robot car, while EP upgrades S1 to infrared sensors and an integrated instruction-based Python API. Such updates lead to different observability and controllability features of RoboMaters, and could potentially affect the quality of P's inputs.

5 Evaluation

In this section, we present the experiments based on our RM-Testing platform. The experiments aim to study the following two research questions.

*RQ*1. Do the selected approaches work in a physical environment as well as in a simulated environment, in preventing a subject context-aware system from failing?

*RQ*2. How do the recognized three factors, namely scenario setting, environmental model, and physical platform, lead to the differences between the selected approaches' performance in a simulated environment and that in a physical environment?

Notice that in the following evaluation, we measure an input validation approach's performance by its effectiveness and usefulness. But our objective is to compare

the differences of the approach's performance in different environments, not to validate its effectiveness and usefulness in these environments.

5.1 Evaluation Design

Scenario. We design different scenarios for investigating the two research questions. For RQ1, we use a scenario based on static urban road network, which consists of straight roads and intersections, simulating the real world. To simplify the scenario, we design the map according to the following three principles: 1) all roads have the same width; 2) all roads are on the same plane; 3) all intersections are one of three pre-defined types (as shown in Fig.4). We also specify the starting point and the end point in the map. The choice of the starting and the end point guarantees the uniqueness of the correct route for the robot car. As a result, the robot car must turn in the correct direction at each intersection in order to reach the end point. We put up road signs on the ground to indicate the correct direction in every interaction. The map we design is shown in Fig.5.

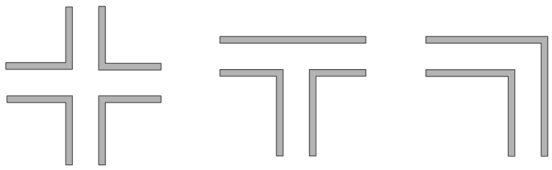


Fig.4. Predefined intersection types [18].

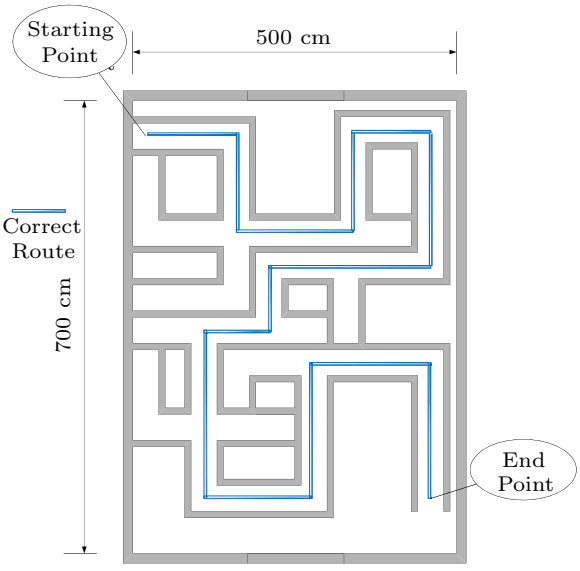


Fig.5. A urban road network for RQ1 [18].

In RQ2, we design a hierarchical maze, as shown in Fig.6, to study the impacts of the three recognized factors. The maze consists of three right turns and three left turns. We also put up a road sign on the opposite fence to indicate the correct direction at every intersection. The maze also guarantees the uniqueness of the correct route for the robot car.

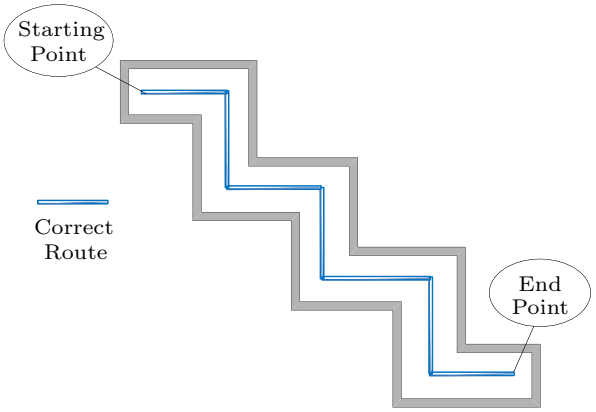


Fig.6. A hierarchical maze for RQ2.

Task. In the initial states of both scenarios, the robot car is placed at the starting point. Then the autopilot program controls the robot car toward the end point. The program should adjust the car's actions in real time based on the environmental data sensed by the range sensors and the camera. To successfully complete the task, the autopilot program has to consider the following two objectives.

• *No Collision.* The autopilot program should avoid the car crashing into any obstacle. The program should first analyze the car's relative position on the road, and keep a distance from all surrounding obstacles while moving towards the end point.

• *No Wrong Turning.* The autopilot program should make sure that the car turns in the correct directions at all intersections. The program uses a deep-learning based image recognition module to identify the road signs and the correct directions at intersections.

An autopilot program's execution is considered "failed" if any of the two objectives is violated, and considered successful if the robot car reaches the end point without failure. The program's quality is measured by "success rate", which is the ratio of the number of success rounds to the number of total rounds.

Subject Context-Aware System. Conducting experiments in both the simulated and physical environments requires a full understanding of the subject program under test. The subject system we use in the experiments

is designed and developed by a well-trained senior undergraduate student. In a simulated environment free from uncertainty, the subject program could control the robot car to reach the end point with a high success rate. We will discuss the limitation of subject selection in Subsection 5.5.

Input Validation. When we put the subject autopilot program in the physical environment, the success rate for the car to reach the end point is lowered because the program suffers from uncertainty. As discussed before, input validation can be used to help the program better cope with uncertainty. The three aforementioned input validation approaches are used as follows.

• We use constraints checking to detect and repair the consistency error of sensor data, improving its reliability.

• We use invariant checking to detect and repair the car's abnormal state, avoiding collision as much as possible.

• We use DL model input pruning to improve the accuracy of image recognition.

We select these representative approaches for the following two reasons. One concerns the motivation of this work that demonstrates the different performance of an input validation approach in the simulated and the physical environment. In this work, we are not trying to compare the effectiveness of different approaches, but trying to compare the same approach's performance in different environments. In other words, the expected conclusion of this empirical work is "the performance of approach A in a physical environment is (not) different from that in a simulated environment", instead of "approach A is more effective than approach B, in the simulated/physical environment". The three selected approaches focus on three different aspects that are important for validating context-aware systems' inputs. Another reason concerns the implementation of the selected approaches. Since few relevant input validation approaches are evaluated in a physical environment, and none of these are directly applicable to our RMTesting platform, we have to implement the selected approaches on the platform. Considering the engineering efforts and the scope of the available approaches, we select three approaches proposed and designed by our colleagues.

Table 1 describes the configurations of input validation approaches in our evaluation. We use checkmark ("✓") to indicate that the concerned approach is enabled in the corresponding configuration.

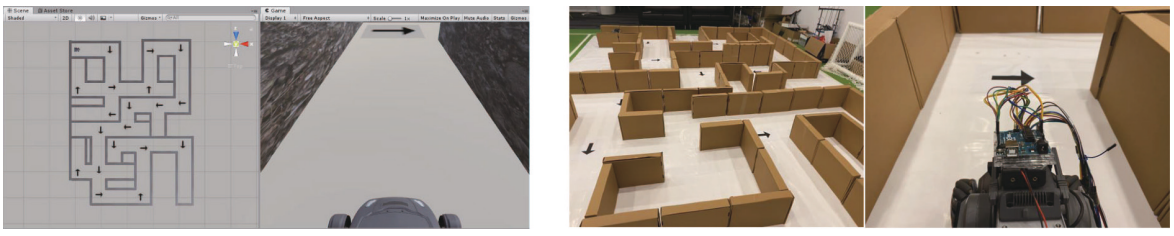
Table 1. Configurations of Input Validation Approaches [18]

Configuration ID	Constraints Checking	Invariant Checking	DL Model Input Pruning
1			
2			✓
3		✓	
4		✓	✓
5	✓		
6	✓		✓
7	✓	✓	
8	✓	✓	✓

5.2 Evaluation Setup

We construct the scenario based on the aforementioned map in the physical and the simulated environments, respectively (as shown in Fig.7). Configurations like object scale, car behavior, range sensor installation position, and camera angle in these environments are the same.

Physical Environment. For the road map in RQ1, we construct the physical scenario on the flat ground with 500 cm × 700 cm. We use some white papers to cover the ground for two considerations. On the one hand, papers can make the road as flat as possible to prevent small potholes on the ground from affecting the car. On the other hand, the original color of the ground is inconsistent, and white papers help to reduce the impact on image recognition. We also use paper boxes as fences on both sides of the road and post a road sign at every intersection to indicate the correct direction.



(a) (b)

Fig.7. Evaluation in different environments [18]. (a) In the simulated environment. (b) In the physical environment.

For the maze in RQ2, we construct the physical scenario on the wooden ground. We use foam boards as fences on both sides of the roads and post road signs at all six interactions. Note that the physical environment in RQ2 is newly built for this journal extension since our previous physical environment for RQ1 has been removed. As a result, the experimental data in RQ1 cannot be directly compared with the data we derived in RQ2.

Simulated Environment. We construct the simulated scenario with the Unity engine[5]. To simulate uncertainty such as random errors and mechanical deviation, we inject several random values based on the characteristics of the physical environment [38]: 1) we add a normally distributed random value to the reading of the range sensor to simulate the random error; 2) we add a large value N to the reading of range sensor with a small probability P to simulate sudden change due to transient sensor failure; 3) we add a normally distributed random value to the steering angle of the car to simulate the mechanical deviation.

5.3 Evaluation Procedure

We conduct the experiments on a laptop with AMD Ryzen 7 4800U CPU @1.8 GHz and 16 GB RAM. For the experiments in the physical environment, we deploy our RM-Testing platform on the aforementioned laptop. For the experiments in the simulated environment, we run the subject program in the Unity engine on the laptop.

To answer research question RQ1, we conduct two groups of experiments. We first perform a broad comparison of different configurations' success rate, denoted as exp-1. Specifically, we run the experiments in the simulated environment of all eight configurations 50 times, and run the experiments in the physical environment with configurations 1 and 8 (i.e., the configurations with all approaches disabled and all approaches enabled, respectively) 50 times to observe the overall performance of these three approaches. We report the success rate in exp-1. We also compare the success rate in the simulated environment and the physical environment.

To answer research question RQ2, we conduct three groups of experiments, each of which studies the impact of one factor. The first group of experiments, which is denoted as exp-2, studies the impact of scenario settings. Specifically, we run the experiments of configurations 4, 6, 7, and 8 (i.e., the configuration with all approaches enabled, and the configurations with each of the approaches disabled) in both the simulated and the physical environment. We run the experiments of each configuration 30 times, and report the percentages of rounds that reach different intersections.

The second group of experiments in RQ2, which is denoted as exp-3, studies the impact of physical platform. Specifically, we additionally refit our RM-testing platform with a RoboMaster EP robot car (an upgraded version of the previous S1 robot car). We run the experiments of configurations 4, 6, 7, and 8 (similar to exp-2) with EP robot car in the physical environment. We also run the experiments of each configuration 30 times, and report the percentages of rounds that reach different intersections.

The third group of experiments in RQ2, denoted as exp-4, studies the impact of the environmental model. Specifically, we analyze the execution traces collected in exp-2 and exp-3, and investigate different environments' feedback on the program's output. We split the collected execution into individual iterations, each of which consists a pair of $\{o_P(n)$ (the program's output of the n-th iteration), $i_P(n+1)$ (the program's input of the $(n+1)$-th iteration)$\}$. Then we use linear regression to derive linear models that describe the environmental logic behind different environments. By comparing the derived linear models of the simulated and the physical environment, we could partly answer whether our simulated environment describes the physical world precisely. Considering various actions a robot car could perform, we only focus on the "moving forward" action in the experiments. We measure the relationship between the action's parameter (i.e., the time that the robot car should move forward) and the action's actual effect (i.e., the average speed that the robot car travels following the action).

An overview of the experiments is shown in Table 2, which follows the experimental process introduced by Wohlin *et al.* [39]. In each experiment, we select the experimental factor according to the cause-effect relationship [39], strictly fix other independent variables, run the subject program on a robot car to help it reach the end point in the object scenario, and observe the dependent variable to evaluate the effect of the factor. We will discuss more details about threats to the validity of the experiments in Subsection 5.5.

[5] https://unity.com, Sept. 2021.

Table 2. Overview of Experiments' Designs

Exp.	Subject	Object	Variable			#Tests
			Factor (Treatments)	Fixed Independent	Dependent	
exp-1	Autopilot program P v1	Scenario in Fig.5	Environment (simulated, physical)	Initial state, object scale, etc.	Success rate	50
exp-2	Autopilot program P v2	Scenario in Fig.6	Scenario settings (1–6 intersections), Environment (simulated, physical)	Initial state, object scale, etc.	Success rate	30
exp-3	Autopilot program P v2	Scenario in Fig.6	Physical platform (RoboMaster S1, RoboMaster EP)	Initial state, scenario settings, etc.	Success rate	30
exp-4	Autopilot program P v2	Scenario in Fig.6	Environmental model (physical, simulated with light uncertainty, simulated with severe uncertainty)	Action type, etc.	Speed/duration	/

Note: Column #Tests is the number of rounds for each configuration group, which is defined in Table 1. The #Tests of exp-4 is not a number because exp-4 considers only the "moving forward" action in the traces of exp-2 and exp-3.

5.4 Evaluation Results and Analyses

5.4.1 *RQ1: Difference Between Performance in Different Environments*

Table 3 gives an overview of the results of exp-1 on the success rate by the eight groups under comparison, in which each group is executed for 50 rounds. When all three approaches are disabled, the car never reaches the end point. We think the main reason for this phenomenon is that the end point is far away from the starting point; therefore, the accumulated deviation causes the car to deviate from the correct route. With all three approaches enabled, the success rate increases to 82%, indicating that these approaches can effectively prevent the car from failing in the simulated environment.

Table 3. Exp-1: Success Rate in Simulated Environment by Configurations [18]

Config. ID	#Success	%Success (%)
1	0	0
2	0	0
3	14	28
4	7	14
5	0	0
6	1	2
7	36	72
8	41	82

Note: Column #Success is the number of successful rounds. Column %Success is the ratio of successful rounds to total rounds (i.e., 50).

To demonstrate the performance of each approach, we also calculate the success rate from the perspective of each approach. Table 4 presents the number of success rounds (#Success) out of 200 rounds with each approach enabled or disabled. We find that constraints checking increases the success rate by 28.5%, and invariant checking increases the success rate by 48.5%, which shows these two approaches' performance. However, DL model input pruning decreases the success rate by 2.0%. We think the main reason is that the accuracy of the original model has already reached a high level (about 94.8%); therefore, the failure caused by prediction errors is rare compared with random errors or mechanical deviations. Therefore, this metric cannot show the performance of DL model input pruning. On the other hand, decreasing by 2.0% is within a reasonable error range.

Table 4. Exp-1: Success Rate Comparison in the Simulated Environment [18]

Approach	#Success		
	Disabled	Enabled	Δ
Constraint checking	21(10.5%)	78(39.0%)	+57(+28.5%)
Invariant checking	1(0.5%)	98(49.0%)	+97(+48.5%)
DL model input pruning	53(26.5%)	49(24.5%)	−4(−2.0%)

Note: Column disabled (enabled) is the number of successful rounds with the corresponding approach disabled (enabled). Column Δ is their difference. The number in brackets is the ratio of the number of successful rounds to the number of total rounds (i.e., 200).

Considering the outlier of DL model input pruning, we further investigate its performance by analyzing the accuracy. For all images in the experimental group with DL model input pruning enabled, we predict their labels with the original deep learning model (DL model input pruning is disabled now) and then observe the change of the accuracy. As shown in Table 5, this ap-

proach helps increase the accuracy by 1.6%. Since the base was high, we think it has been a big improvement.

Table 5. Exp-1: Accuracy of Image Classification in the Simulated Environment [18]

Config. ID	#Samples	Accuracy (%)	
		Disabled	Enabled
2	190	91.9	90.5(−1.4%)
4	371	98.4	99.2(+0.8%)
6	302	86.4	90.7(+4.3%)
8	706	97.6	99.0(+1.4%)
Total	1 569	94.8	96.4(+1.6%)

Note: Column #Samples is the number of image samples in each group. Column disabled (enabled) is the accuracy of image classification with DL model input pruning disabled (enabled). The number in the brackets is their difference.

Then we study the performance of input validation in the physical environment. Unfortunately, even with all three approaches enabled, the robot car fails to reach the end point in all of its 50 executions. The major reason for this zero success rate is the system's low accuracy in recognizing the road signs. On the one hand, recognizing the images in the physical environment is much more difficult than in the simulated environment. We try our best to achieve an 80% recognition accuracy after tuning the model for several days. On the other hand, even the accuracy of sign recognition reaches 90%, and the probability for the robot car to reach the end point is barely over 20%, which could be further lowered by other uncertain factors.

As a result, we measure the proportions of the robot car reaching the fifth and the sixth intersections, instead of its success rate reaching the end point. The results are shown in Table 6. With all three approaches enabled, reaching the 5th and the 6th intersections increases by 50% and 24%, respectively. The results indicate that these approaches can also effectively prevent the car from failing in the physical environment.

Table 6. Exp-1: Success Rate in the Physical Environment [18]

Config ID	Percentage of Reaching the 5th Intersection	Percentage of Reaching the 6th Intersection
1 (all disabled)	4%	0%
8 (all enabled)	54%(+50%)	24%(+24%)

As mentioned above, the car passing all the 15 intersections and reaching the end point reaches 82% in the simulated environment, while its proportion reaching the 6th intersection is 24% in the physical environment. Although these approaches are effective in both environments, we mainly focus on the difference between (approaches' performance in) different environments. For RM-Testing with RoboMaster S1, the performance of these input validation approaches in the physical environment is not so significant as that in the simulated environment.

Therefore, we answer research question RQ1 as follows.

*Answer to RQ*1. The performance of input validation approaches in a simulated environment is significantly different from that in a physical environment. Specifically, the selected approaches could improve the subject context-aware system's success rate on a complicated task (i.e., passing 13 interactions) by 82% in a simulated environment, while only improving that on an easy task (i.e., passing five interactions) by 50% in a physical environment [18].

5.4.2 *RQ2: Impact of Scenario Setting, Environmental Model, and Physical Platform*

We first study the impact of scenario setting in distinguishing a physical environment and a simulated environment (exp-2). The results are presented in Table 7. We run the robot car in each environment for 30 rounds, and compare the number of rounds that the car passes each intersection in the two environments. We use the decrease of the number of successful rounds when the approach is disabled to measure the performance of a concerned input validation approach (indicated by "$-i$" following the reported number of rounds). Basically, the results show that scenario setting has different impacts on the selected input validation approaches' performance in different environments. For the physical environment, the selected approaches' performance increases as the growth of the number of intersections the robot car passes (avg. −1.67 at intersection 1, and avg. −9 at intersection 6). This follows an intuitive observation that environmental uncertainty accumulates along with a context-aware system's execution. We conjecture that the more extensive the uncertainty a context-aware system faces, the more effective the input validation would be. We leave this issue as our future work. When it comes to the simulated environment, the decrease of rounds remains stable when the number of intersections grows (avg. −1 at intersection 1, and avg. 0 at intersection 6). This echoes our previous finding in RQ1 that the simulated environment does differ from the physical environment in terms of illustrating the performance of input validation approaches.

Table 7. Exp-2: Comparison Between Physical and Simulated Environments with Different Scenario Settings

Environment	Group ID	#Rounds Passing the n-th Intersection					
		$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
Physical with EP	8 (✓,✓,✓)	29	27	27	23	18	17
	4 (×,✓,✓)	28(−1)	25(−2)	24(−3)	22(−1)	15(−3)	9(−8)
	6 (✓,×,✓)	26(−3)	25(−2)	22(−5)	21(−2)	11(−7)	9(−8)
	7 (✓,✓,×)	28(−1)	21(−6)	20(−7)	15(−8)	12(−6)	6(−11)
	avg(4, 6, 7)	27.3(−1.7)	23.7(−3.3)	22.0(−5.0)	19.3(−3.7)	12.7(−5.3)	8.0(−9.0)
Simulated	8 (✓,✓,✓)	30	25	21	18	16	11
	4 (×,✓,✓)	28(−2)	22(−3)	16(−5)	14(−4)	13(−3)	8(−3)
	6 (✓,×,✓)	30 (0)	25(0)	20(−1)	18 (0)	14(−2)	11(0)
	7 (✓,✓,×)	29(−1)	26(+1)	22(+1)	20(+2)	15(−1)	14(+3)
	avg(4,6,7)	29.0(−1.0)	24.3(−0.7)	19.3(−1.7)	17.3(−0.7)	14.0(−2.0)	11.0(0.0)

Note: The marks in column Group ID denote whether each approach (i.e., constraint checking, invariant checking, and DL model input pruning, respectively) is enabled or not. The number in the colored brackets is the decrease of successful rounds compared with the group that all the approaches are enabled (i.e., group 8).

Then we compare the selected approaches' performance with the different physical platforms (exp-3), as shown in Table 8. The result shows that RoboMaster S1 fails to reach the 5th and the 6th intersections, which can be regarded as highly complex scenarios, while RoboMaster EP passes them with a high probability. The major reason for RoboMaster S1's failure is the significant uncertainty in its interaction with the physical environment, as we explained in RQ1 and exp-2. Nevertheless, in the scenarios whose complexities are within both the robot cars' capability (e.g., both S1 and EP can pass the 1st and the 2nd intersections with a probability of more than 50%), the input validation approaches show better performance on S1. We conjecture that this difference results from different extents of environmental uncertainty suffered by EP and S1. Compared with S1, EP is equipped with better sensors (standard infrared range sensors for EP, and self-resembled ultra-sonic range sensors for S1) and a better control interface (manufacturer-provided APIs for EP, and FPV controller for S1). As a result, for the physical platform that suffers significant uncertainty (e.g., RoboMaster S1), the concerned input validation approaches help a lot to alleviate the uncertainty and reduce the abnormal rate. For the physical platform that suffers slight uncertainty (e.g., the simulated robot car), those approaches are less effective since environmental uncertainty does not affect the quality of sensor read-

Table 8. Exp-3: A Comparison Between the Physical Environments with Different Physical Platforms

Environment	Group ID	#Rounds Passing the n-th Intersection					
		$n=1$	$n=2$	$n=3$	$n=4$	$n=5$	$n=6$
Physical with S1	8 (✓,✓,✓)	26	19	5	1	0	0
	4 (×,✓,✓)	22(−4)	12(−7)	2(−3)	0(−1)	0(0)	0(0)
	6 (✓,×,✓)	24(−2)	14(−5)	4(−1)	0(−1)	0(0)	0(0)
	7 (✓,✓,×)	19(−7)	16(−3)	5 (0)	3(+2)	0(0)	0(0)
	avg(4, 6, 7)	21.7(−4.3)	14.0(−5.0)	3.7(−1.3)	1.0(0.0)	0.0(0.0)	0.0(0.0)
Physical with EP	8 (✓,✓,✓)	29	27	27	23	18	17
	4 (×,✓,✓)	28(−1)	25(−2)	24(−3)	22(−1)	15(−3)	9(−8)
	6 (✓,×,✓)	26(−3)	25(−2)	22(−5)	21(−2)	11(−7)	9(−8)
	7 (✓,✓,×)	28(−1)	21(−6)	20(−7)	15(−8)	12(−6)	6(−11)
	avg(4, 6, 7)	27.3(−1.7)	23.7(−3.3)	22.0(−5.0)	19.3(−3.7)	12.7(−5.3)	8.0(−9.0)

Note: The marks in column Group ID denote whether each approach (i.e., constraint checking, invariant checking, and DL model input pruning, respectively) is enabled or not. The number in the colored brackets is the decrease of successful rounds compared with the group that all the approaches are enabled (i.e., group 8).

ings severely. However, despite this difference between S1 and PE, the results of exp-3 suggest that physical platform does affect the input validation approaches' performance.

Last but not least, we study how the environmental model affects the behavior of the subject program in its running environment (exp-4). Fig. 8 compares the relationship derived from the robot car's execution traces in the physical environment and the simulated environment. The results show that the simulated and physical environments have different internal environmental models that react to a context-aware system's actions. Changing the extent of introduced uncertainty would make the derived relationship different even with the same simulated environment. As a result, we conjecture that uncertainty could not be the only major factor contributing to the difference between the simulated and physical environments. However, the exploration of the other root factors needs future research efforts.

Therefore, we answer research question RQ2 as follows.

*Answer to RQ*2. Scenario setting, physical platform, and environmental model have different impact on the selected input validation approaches' performance in different environments. The impact of scenario setting is stale in the simulated environment, compared with the physical environment. The physical platform has a significant influence on the behavior of a context-aware system, which could partly overturn the selected approaches' performance. The environmental model varies in different environments, especially when the extent of uncertainty changes.

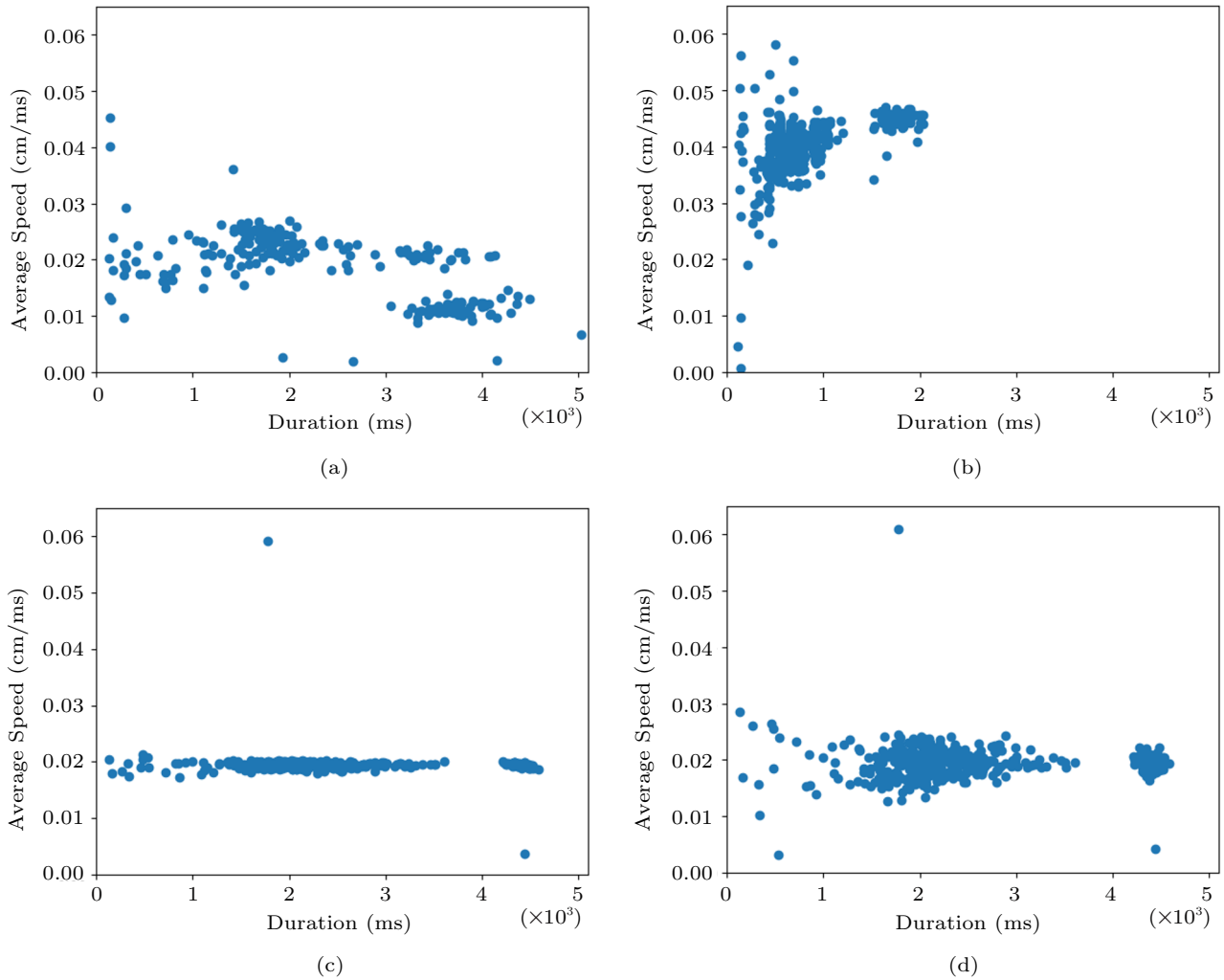


Fig.8. Exp-4: comparison between the action/effect relationship derived from different environments. (a) Physical environment with RoboMaster S1. (b) Physical environment with RoboMaster EP. (c) Simulated environment with light uncertainty. (d) Simulated environment with severe uncertainty.

5.5 Threats to Validity and Discussions

We present our discussions on those major threats from the following three aspects: the subject and object of the experiments, the variables of the experiments, and the number of tests of the experiments.

5.5.1 Subjects and Objects

Threat. A major threat is that the autopilot program presented in Algorithm 1 is the only investigated subject program.

Discussion. Conducting experiments in both the simulated environment and the physical environment requires a full understanding of the testing program, which restricts our choice of potential subject programs. However, we believe it can represent real-world context-aware systems for the following reasons. First, the autopilot program of a self-driving car is a typical context-aware system [19, 20] and is widely used as the subject in existing studies [38, 40]. Our program also represents and integrates various kinds of autopilot-related context-aware services, including collision avoidance, disengagement detection, and road sign recognition. Second, we have tried various combinations of configurations and hardware for the subject. For example, we use two versions of the program (v1 and v2, as shown in Table 2) and ensure each version works correctly in the corresponding scenario. We implement the autopilot programs for multiple robot cars (RoboMaster S1, RoboMaster EP, and a simulated robot car), and run the program with multiple configurations (eight configuration groups, as shown in Table 1). For objects, we have tried two scenarios (as shown in Fig.5 and Fig.6), among which the second one can provide multiple levels of difficulty by controlling the number of intersections (as shown in Table 7 and Table 8).

5.5.2 Variables

Threat. A major threat is the difference between the number of configuration groups in the simulated environment and that in the physical environment in exp-1.

Discussion. As we discussed above, we use less configuration groups in the physical environment because the autopilot program failed to lead the robot car to the end point. Nevertheless, we believe our current experiments could indicate the difference between selected approaches' performance in the simulated and physical environments. We answer RQ1 with only the results of their common configuration groups. The different configurations also validate the differences between a context-aware system's execution in a simulated environment and that in a physical environment. This echoes our finding that context-aware systems based studies should be evaluated in the physical environment.

5.5.3 Number of Conducted Tests

Threat. A major threat is that we run the experiments for only 30 or 50 rounds for each configuration group of each experiment.

Discussion. The main reason for this issue is the high cost of conducting experiments in the physical environment [17]. However, considering that we run the experiments for multiple configuration groups, we run many rounds in total (100, 120, and 120 rounds in the physical environment in exp-1, exp-2, and exp-3, respectively).

6 Related Work

In this section, we will discuss the dynamic physical environment that a context-aware system usually faces (related to background), introduce empirical study designs (i.e., whether evaluating in a simulated or a physical environment) of some input validation approaches for context-aware systems (related to motivation), present some studies concerning the difference of the simulated and the physical environments (related to RQ1), and finally discuss some studies related to the three recognized factors (related to RQ2).

6.1 Context-Aware Systems in Dynamic Physical Environment

Different from a cyber laboratory environment that is closed and static, the context-aware system is open and dynamic [4, 7, 9, 36], which means the execution environment remains unknown for developers. Dynamic environmental conditions may compromise the ability of adaptation and decision-making of context-aware systems [9]. Existing approaches try to mitigate the effect of environmental dynamics [9, 24, 25, 38] mainly by enhancing the adaptation process of the systems.

Context is of central importance when the system is running in the dynamic physical environment. Context-aware systems need to keep track of context [36]. The notion and the representation of context vary in the literature. Dourish [36] summarized four assumptions of context: a form of information, delineable, stable, and separable from activity. Dey *et al.* [41] defined context as "any information that characterizes a situation

related to the interaction between humans, applications, and the surrounding environment". Matalonga and Travassos [35] abstracted context as context variables and inputs from sensors. Sama *et al.* [4] defined the propositional context variable as "the abstract representation of relational expression over sensed context variables". We apply a definition similar to [4, 35] because of the similarity of scenarios.

Paden *et al.* [19] and Badue *et al.* [20] divided an autopilot program of a self-driving car, a typical context-aware system running in a dynamic physical environment, into two parts: the perception system and the decision-making system. The perception system is responsible for sensing the dynamic environment to obtain the latest information, while the decision-making system makes decisions in response to environmental changes [20]. A self-driving car may face various and dynamic traffic situations, and thus the Behavior Selector, a subsystem of the decision-making system, is responsible for selecting actions by some techniques according to the current state [20].

6.2 Empirical Study Designs of Input Validation Approaches for Context-Aware Systems

We survey empirical study designs of some input validation approaches for the context-aware system. The overall results are shown in Table 9.

Table 9. Survey on Literature About How They Evaluate the Approaches

LIT	Studied CAS	P?	S?
[5]	Robot car, Avatar simulator	✓	✓
[6]	RFID-based warehouse system		✓
[7]	NAO robot, UAV	✓	✓
[8]	Self-driving car		✓
[9]	Intelligent vehicle system		✓
[10]	Smart meter/medical devices		✓
[11]	Location-based phone-adaptor		✓
[12]	Syllabus management system		✓
[13]	UML design tools		✓
[14]	Table datasets		✓
[15]	Self-driving car		✓
[16]	Self-driving car		✓

Note: Column LIT is the literature number. Column P? (S?) means whether this study evaluated the approach in the physical environment (in the simulated/trace-based environment).

Constraint Checking. Nentwich *et al.* [12] proposed xlinkit for repairing inconsistent XML documents. This framework builds on an incremental checking model. Egyed [13] focused on repairing inconsistency errors in UML models. These studies [12, 13] evaluated the approaches with pre-collected or generated context data.

Xu *et al.* [6] proposed two strategies in efficient checking inconsistent context, namely partial constraint checking strategy (denoted as PCC) and entire constraint checking strategy (denoted as ECC). Both the PCC and the ECC approaches were evaluated in the simulated environments using real context data.

Invariant Checking. Invariant checking enables programs to detect potential abnormal states at runtime. Xu *et al.* [5] monitored runtime errors for an application and related them to responsible defects in the application. Qin *et al.* [7] explored multi-invariant detection based on context-based trace grouping. These studies [5, 7] evaluated the approaches in both the simulated and the physical environments.

Besides, Ramirez *et al.* [9] proposed an approach that discovers combinations of environmental conditions to trigger specification-violating behaviors. Aliabadi *et al.* [10] explored mining dynamic system properties around time. Wang *et al.* [11] proposed identifying program points where the system's behavior may be affected by context changes. These studies [9–11] evaluated the approaches in the simulated environments using generated context data.

DL Model Input Pruning. Input validation for the deep learning model could greatly improve the performance of DL models in terms of their accuracy in predicting. Chu *et al.* [14] focused on data cleaning for more qualified training data. Pei *et al.* [15] converted the corner-case generation problem to the joint optimization problem. Tian *et al.* [16] proposed a testing tool for detecting the abnormal state of DNN-driven vehicles that can potentially lead to crashing. Wang *et al.* [8] tracked each input's interpretation for estimating its validity. These studies [8, 14–16] evaluated the approaches using static datasets.

As discussed above, most related studies evaluated the proposed approaches in the simulated environments, and few conducted experiments in the physical environments.

6.3 Difference of Simulated and Physical Environments

As far as we know, few studies focused on the difference of evaluation in simulated and physical environments for context-aware systems. Magnusson [42] invited five fighter pilots to fly the same mission in both a simulator and a real flight, examining the difference

of their reactions. Bishop and Rohrmann [43] compared respondents' cognitive reactions to a real urban park environment and a simulated one.

Although these studies and ours all discuss the difference between simulated and physical environments, they mainly studied from the perspectives of psychology, human-robot interaction, or simulated-physical integration, while we focus on the evaluation of input validation approaches for the context-aware system.

6.4 Potential Factors Leading Difference

Many studies explicitly or implicitly concerned the effect of the three recognized potential factors.

Scenario Setting. Fredericks *et al.* [24] used utility functions to guide the adaptation process of context-aware systems and provide assurance when the scenario changes. Ramirez *et al.* [9] proposed an approach to search for specific combinations of environmental conditions that violate the requirements. Jiang *et al.* [25] tried to automatically infer system invariants that capture the temporal and spatial aspects, which can help reduce the failure rate in a complex scenario. The conditions can be further used to generate test cases. These studies tried to make the context-aware system less affected by the scenario setting factor and behave safely in an unanticipated scenario.

Environmental Model. Qin *et al.* [44] presented a sampling-based approach for testing context-aware systems. They identified "infinite reaction loop" and "uncertain interaction" as the major issues that make a context-aware system easily error-prone, echoing our environmental model factor and physical platform factor, respectively. Weyns *et al.* [45] introduced a notation for describing multiple interacting MAPE loops [46]. In the constituent parts of their self-adaptive system, "environment" refers to an external component that corresponds to both non-controllable software and hardware entities [45]. The insight is similar to our environmental model factor.

Physical Platform. Ramirez *et al.* [37] reported a taxonomy of uncertainty for dynamically adaptive systems. They defined runtime uncertainty as that that occurs from the interaction between the system and its unpredictable environment, and then introduced some related work for managing it. Xu *et al.* [6] presented a partial constraint checking approach for pervasive computing to prevent the system from being seriously affected by the sensor noise. Yang *et al.* [38] tried to verify self-adaptive applications through modeling adaptation logic and environmental constraints with uncertainty explicitly considered. All these studies concerned the interaction between context-aware systems and the environments. They were dedicated to making context-aware systems less affected by the physical platform factor.

For clarity, the aforementioned studies explicitly or implicitly concerned part of the three factors mainly for proposing approaches to control or limit them, while we focus on investigating the impact of the factors. There is an essential difference between existing studies and ours.

7 Conclusions

In this article, we built a testing platform for context-aware systems to study whether different environments (i.e., physical or simulated) could lead to different performance of input validation approaches. The experimental results showed that the performance of three up-to-date input validation approaches (i.e., constraint checking, invariant checking, and DL model input pruning) in the simulated environment (improving the task success rate by 82% compared with the system without these approaches) does differ from their performance in the physical environment (improving the task success rate by 50%). Based on an execution model of the context-aware system called PEIM, we also recognized three factors that may affect the performance of input validation approaches, namely scenario setting, environmental model, and physical platform. The experimental results showed that these factors have different impacts on the performance of input validation approaches in different environments, among which the physical platform makes the most significant influence.

Our work still has room for improvement. It currently only relies on one type of the context-aware systems (i.e., autopilot program). The involved environmental scenarios designed by ourselves might lack diversity. Although our experimental results demonstrated the selected approaches' different performance between the simulated and the physical environments under these settings, more concerned context-aware systems and more environments could make our conclusions apply to a broader scope.

Besides, the work also brings new research opportunities. Developing reliable context-aware systems is well-evidenced to be a difficult task in the community of software engineering. Our work can be regarded as an

explanation of how the low-quality inputs could harness the reliability of a context-aware system. We observed a context-aware system's different behaviors after tempering different factors. Specific approaches could be proposed to alleviate the negative impact of uncertain inputs toward these factors. This needs further research and validation, and we keep it as future work.

References

[1] Lü J, Ma X, Huang Y, Cao C, Xu F. Internetware: A shift of software paradigm. In *Proc. the 1st Asia-Pacific Symposium on Internetware*, October 2009, Article No. 7. DOI: 10.1145/1640206.1640213.

[2] Sama M, Elbaum S, Raimondi F, Rosenblum D S, Wang Z. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Transactions on Software Engineering*, 2010, 36(5): 644-661. DOI: 10.1109/TSE.2010.35.

[3] Qin Y, Xu C, Chen Z, Lü J. Software testing for cyber-physical systems suffering uncertainty. *SCIENTIA SINICA Informationis*, 2019, 49(11): 1428-1450. DOI: 10.1360/N112018-00305. (in Chinese)

[4] Sama M, Rosenblum D S, Wang Z, Elbaum S. Model-based fault detection in context-aware adaptive applications. In *Proc. the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2008, pp.261-271. DOI: 10.1145/1453101.1453136.

[5] Xu C, Cheung S C, Ma X, Cao C, Lu J. Adam: Identifying defects in context-aware adaptation. *Journal of Systems and Software*, 2012, 85(12): 2812-2828. DOI: 10.1016/j.jss.2012.04.078.

[6] Xu C, Cheung S C, Chan W K, Ye C. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology*, 2010, 19(3): Article No. 9. DOI: 10.1145/1656250.1656253.

[7] Qin Y, Xie T, Xu C, Astorga A, Lu J. CoMID: Context-based multiinvariant detection for monitoring cyber-physical software. *IEEE Transactions on Reliability*, 2019, 69(1): 106-123. DOI: 10.1109/TR.2019.2933324.

[8] Wang H, Xu J, Xu C, Ma X, Lu J. DISSECTOR: Input validation for deep learning applications by crossing-layer dissection. In *Proc. the 42nd ACM/IEEE International Conference on Software Engineering*, Oct. 2020, pp.727-738. DOI: 10.1145/3377811.3380379.

[9] Ramirez A J, Jensen A C, Cheng B H, Knoester D B. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *Proc. the 26th IEEE/ACM International Conference on Automated Software Engineering*, November 2011, pp.568-571. DOI: 10.1109/ASE.2011.6100127.

[10] Aliabadi M R, Kamath A A, Gascon-Samson J, Pattabiraman K. ARTINALI: Dynamic invariant detection for cyber-physical system security. In *Proc. the 11th Joint Meeting on Foundations of Software Engineering*, Sept. 2017, pp.349-361. DOI: 10.1145/3106237.3106282.

[11] Wang Z, Elbaum S, Rosenblum D S. Automated generation of context-aware tests. In *Proc. the 29th International Conference on Software Engineering*, May 2007, pp.406-415. DOI: 10.1109/ICSE.2007.18.

[12] Nentwich C, Capra L, Emmerich W, Finkelsteiin A. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2002, 2(2): 151-185. DOI: 10.1145/514183.514186.

[13] Egyed A. Fixing inconsistencies in UML design models. In *Proc. the 29th International Conference on Software Engineering*, May 2007, pp.292-301. DOI: 10.1109/ICSE.2007.38.

[14] Chu X, Morcos J, Ilyas I F, Ouzzani M, Papotti P, Tang N, Ye Y. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 31-June 4, 2015, pp.1247-1261. DOI: 10.1145/2723372.2749431.

[15] Pei K, Cao Y, Yang J, Jana S. DeepXplore: Automated whitebox testing of deep learning systems. In *Proc. the 26th Symposium on Operating Systems Principles*, October 2017, pp.1-18. DOI: 10.1145/3132747.3132785.

[16] Tian Y, Pei K, Jana S, Ray B. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proc. the 40th International Conference on Software Engineering,* May 27–June 3, 2018, pp.303-314. DOI: 10.1145/3180155.3180220.

[17] Luo C, Goncalves J, Velloso E, Kostakos V. A survey of context simulation for testing mobile context-aware applications. *ACM Computing Surveys*, 2020, 53(1): Article No. 21. DOI: 10.1145/3372788.

[18] Chen J, Qin Y, Wang H, Xu C. Simulated or physical? An empirical study on input validation for context-aware systems in different environments. In *Proc. the 12th Asia-Pacific Symposium on Internetware*, Nov. 2021, pp.146-155. DOI: 10.1145/3457913.3457919.

[19] Paden B, Čáp M, Yong S Z, Yershov D, Frazzoli E. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 2016, 1(1): 33-55. DOI: 10.1109/TIV.2016.2578706.

[20] Badue C, Guidolini R, Carneiro R V *et al.* Self-driving cars: A survey. *Expert Systems with Applications*, 2021, 165: Article No. 113816. DOI: 10.1016/j.eswa.2020.113816.

[21] Ma N, Gao Y, Li J, Li D. Interactive cognition in self-driving. *SCIENTIA SINICA Informationis*, 2018, 48(8): 1083-1096. DOI: 10.1360/N112018-00028. (in Chinese)

[22] Xi W, Xu C, Yang W, Hong X. How context inconsistency and its resolution impact context-aware applications. *Journal of Frontiers of Computer Science and Technology*, 2014, 8(4): 427-437. DOI: 10.3778/j.issn.1673-9418.1311013. (in Chinese)

[23] Esfahani N, Malek S. Uncertainty in self-adaptive software systems. In *Proc. the International Seminar on Software Engineering for Self-Adaptive Systems*, Oct. 2013, pp.214-238. DOI: 10.1007/978-3-642-35813-5_9.

[24] Fredericks E M, DeVries B, Cheng B H C. Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty. In *Proc. the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, June 2014, pp.17-26. DOI: 10.1145/2593929.2593937.

[25] Jiang H, Elbaum S, Detweiler C. Reducing failure rates of robotic systems though inferred invariants monitoring. In *Proc. the 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, November 2013, pp.1899-1906. DOI: 10.1109/IROS.2013.6696608.

[26] Al-Garadi M A, Mohamed A, Al-Ali A K, Du X, Ali I, Guizani M. A survey of machine and deep learning methods for Internet of Things (IoT) security. *IEEE Communications Surveys & Tutorials*, 2020, 22(3): 1646-1685. DOI: 10.1109/COMST.2020.2988293.

[27] Li Z, Ma X, Xu C, Cao C, Xu J, Lü J. Boosting operational DNN testing efficiency through conditioning. In *Proc. the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, August 2019, pp.499-509. DOI: 10.1145/3338906.3338930.

[28] Li Z, Ma X, Xu C, Xu J, Cao C, Lü J. Operational calibration: Debugging confidence errors for DNNs in the field. In *Proc. the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2020, pp.901-913. DOI: 10.1145/3368089.3409696.

[29] Nentwich C, Emmerich W, Finkelsteiin A, Ellmer E. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 2003, 12(1): 28-63. DOI: 10.1145/839268.839271.

[30] Gehrke J, Madden S. Query processing in sensor networks. *IEEE Pervasive Computing*, 2004, 3(1): 46-55. DOI: 10.1109/MPRV.2004.1269131.

[31] Tong Y, Qin Y, Jiang Y, Xu C, Cao C, Ma X. Timely and accurate detection of model deviation in self-adaptive software-intensive systems. In *Proc. the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, August 2021, pp.168-180. DOI: 10.1145/3468264.3468548.

[32] Ernst M D, Perkins J H, Guo P J, McCamant S, Pacheco C, Tschantz M S, Xiao C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007, 69(1/2/3): 35-45. DOI: /10.1016/j.scico.2007.01.015.

[33] Jiang H, Elbaum S, Detweiler C. Inferring and monitoring invariants in robotic systems. *Autonomous Robots*, 2017, 41(4): 1027-1046. DOI: 10.1007/s10514-016-9576-y.

[34] Bai X, Yang M, Shi B, Liao M. Deep learning for scene text detection and recognition. *SCIENTIA SINICA Informationis*, 2018, 48(5): 531-544. DOI: 10.1360/N112018-00003. (in Chinese)

[35] Matalonga S, Travassos G H. Testing context-aware software systems: Unchain the context, set it free! In *Proc. the 31st Brazilian Symposium on Software Engineering*, September 2017, pp.250-254. DOI: 10.1145/3131151.3131190.

[36] Dourish P. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 2004, 8(1): 19-30. DOI: 10.1007/s00779-003-0253-8.

[37] Ramirez A J, Jensen A C, Cheng B H C. A taxonomy of uncertainty for dynamically adaptive systems. In *Proc. the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, June 2012, pp.99-108. DOI: 10.1109/SEAMS.2012.6224396.

[38] Yang W, Xu C, Liu Y, Cao C, Ma X, Lu J. Verifying self-adaptive applications suffering uncertainty. In *Proc. the 29th ACM/IEEE International Conference on Automated Software Engineering*, September 2014, pp.199-210. DOI: 10.1145/2642937.2642999.

[39] Wohlin C, Runeson P, Höst M, Ohlsson M C, Regnell B, Wesslén A. Experimentation in Software Engineering. Springer, 2000. DOI: 10.1007/978-3-642-29044-2.

[40] Zhang L, Xu C, Ma X, Gu T, Hong X, Cao C, Lu J. Resynchronizing model-based self-adaptive systems with environments. In *Proc. the 19th Asia-Pacific Software Engineering Conference*, December 2012, pp.184-193. DOI: 10.1109/APSEC.2012.62.

[41] Dey A K, Abowd G D, Salber D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 2001, 16(2): 97-166. DOI: 10.1207/S15327051HCI16234_02.

[42] Magnusson S. Similarities and differences in psychophysiological reactions between simulated and real air-to-ground missions. *The International Journal of Aviation Psychology*, 2002, 12(1): 49-61. DOI: 10.1207/S15327108IJAP1201_5.

[43] Bishop I D, Rohrmann B. Subjective responses to simulated and real environments: A comparison. *Landscape and Urban Planning*, 2003, 65(4): 261-277. DOI: 10.1016/S0169-2046(03)00070-7.

[44] Qin Y, Xu C, Yu P, Lu J. SIT: Sampling-based interactive testing for self-adaptive apps. *Journal of Systems and Software*, 2016, 120: 70-88. DOI: 10.1016/j.jss.2016.07.002.

[45] Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka K M. On patterns for decentralized control in self-adaptive systems. *Software Engineering for Self-Adaptive Systems II*, 2013, pp.76-107. DOI: 10.1007/978-3-642-35813-5_4.

[46] Kephart J O, Chess D M. The vision of autonomic computing. *Computer*, 2003, 36(1): 41-50. DOI: 10.1109/MC.2003.1160055.

Jin-Chi Chen is currently a Master student with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. His research interests include self-adaptive systems.

Yi Qin is currently an assistant researcher with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. He received his Ph.D. degree in computer science and technology from Nanjing University, Nanjing, in 2018. His research interests include self-adaptive software systems, and software testing and analysis.

Hui-Yan Wang is an assistant researcher with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. She received her B.S. and Ph.D. degrees in computer science and technology from Nanjing University, Nanjing, in 2015 and 2021 respectively. Her research interests are intelligent software quality assurance, context management, and software analysis and testing. She has published many referred papers in both international software engineering conferences and journals. She is a member of CCF.

Chang Xu is currently a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. He received his Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, in 2008. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He co-authored over 160 peer-reviewed papers and served in technical program committees of various international software engineering conferences. He is a member of ACM, and a senior member of CCF and IEEE.