Simulated or Physical? An Empirical Study on Input Validation for Context-aware Systems in Different Environments

Jinchi Chen
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China
jcchen.nju@gmail.com

Yi Qin
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China
yiqincs@nju.edu.cn

Huiyan Wang
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China
cocowhy1013@gmail.com

Chang Xu
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Nanjing, China
changxu@nju.edu.cn

ABSTRACT

Context-Aware Systems (a.k.a. CASs) integrate cyber and physical space to provide context-aware adaptive functionalities. Building context-aware systems is challenging due to the uncertainty of the real physical environment. Therefore, input validation for context-aware systems plays a significant role in keeping the systems executing safely. Input validation approaches have been proposed to monitor and guard the executions of context-aware systems. However, few of these works (17%, 2 out of 12) evaluated their approaches with a real context-aware system in a real physical environment. In this paper, we study and compare the effectiveness of input validation approaches for context-aware system in both a simulated and a physical environment. We built a testing platform, RM-Testing, based on DJI RoboMaster S1 robot car. We implemented three up-to-date input validation approaches, and evaluated their effectiveness in improving the success rate of the robot car's executions. The results show that the selected input validation approaches are effective in guarantee the safe execution of context-aware systems, which improve the success rate by 82% in the simulated environment, and 50% in the physical environment. However, the effectiveness of these approaches does vary in different environment. Thus, we believe that such CASs-based input validation works should be evaluated in the physical environment to better validate their effectiveness and usefulness.

Internetware'20, May 12–14, 2021, Singapore, Singapore

ACM ISBN 978-1-4503-8819-1/20/11…$15.00
https://doi.org/10.1145/3457913.3457919

KEYWORDS

context-aware systems, self-driving cars, input validation, testing infrastructure

ACM Reference Format:
Jinchi Chen, Yi Qin, Huiyan Wang, and Chang Xu. 2020. Simulated or Physical? An Empirical Study on Input Validation for Context-aware Systems in Different Environments . In *12th Asia-Pacific Symposium on Internetware (Internetware'20), May 12–14, 2021, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3457913.3457919

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 integrate cyber and physical space to provide context-aware adaptive functionalities. These systems continually sense environmental changes, make decisions based on their preprogrammed logic, and then take physical actions to adapt to the sensed changes.

Empirical evidence shows that building context-aware systems is challenging and easily error-prone [15, 24, 25]. One of the reasons is that these programs have to address the complexities incurred during their interaction with the physical environment. Different from traditional programs, context-aware systems mainly execute in the physical environment, which might produce uncertain inputs for the systems. Such uncertain inputs are unpredictable while developing a context-aware system, and they could lead the system to abnormality or failure if are not processed appropriately. Therefore, there is a strong need for validating the input of context-aware systems to prevent them from entering fatal errors (e.g., crashing of Tesla self-driving car [5] and failing of the auto-pilot program of Boeing 737-MAX8 [6]).

Many research efforts have been made to validate the input of context-aware systems from different perspectives, including context consistency constraint checking [11], invariant checking [19],

and deep learning model input pruning [9]. However, few of these works evaluated their proposed approaches with a real context-aware system in a real physical environment. We conducted a primitive empirical study and find that only 17% of these works evaluates their approaches in physical environments (2 out of 12), and others are only evaluated either in a simulated environment 33% (4 out of 12), or with pre-collected execution traces of context-aware systems 50% (6 out of 12). This result echoes a latest survey [4] on self-adaptive systems that one of the major challenges in building adaptive systems is to "providing empirical evidence for the value of self-adaptive (in the real environment)" and avoid validating the research effort with simple example applications in the simulated environment. As such, one could naturally ask two questions that **whether those input-validation-approaches work effectively in a physical environment**, and **whether the difference between a simulated environment and a physical environment affect the effectiveness of those approaches**.

The major challenge to answer these two questions is to build a testing infrastructure in physical world for testing context-aware systems. The testing infrastructure should connect the subject program under test with the physical platform that is capable of sensing the surrounding environment and taking physical actions. Building such a testing infrastructure not only requires efforts on implementing the software modules that support testing and validation approaches, but also require efforts to refit and modify the physical platforms to enrich their sensibility and controllability, as well as building the physical scenario.

What is more, the difference between the simulated environment and the physical environment makes one cannot directly apply those input validation approaches on a real context-aware system in a real physical environment. For example, in related work the application will drop inputs under validity threshold directly to achieve better accuracy perform [9]. However, when it comes to a real robot car recognizing the road sign by deep learning method, the same strategy can lead to failure if all the images cannot reach the fixed threshold.

In this paper, we address these challenges by building a testing platform, RM-Testing, and evaluated three input validation approaches in both the simulated and the real environment. We select autonomous driving, one of the "killer applications" of context-aware systems. We refitted a DJI RoboMaster S1 robot car with additional range sensors to enrich its sensibility towards the surrounding environment. We built a controller module to enable autonomous control of the robot car with subject autopilot programs. We implemented and adapted three input validation approaches, namely *ECC* [11], *CoMID* [19], and *DISSECTOR* [9], within our RM-Testing platform to validate the environmental information pushed to the subject autopilot program.

We conducted extensive experiments to answer the aforementioned two questions. We build a scenario of urban road network in both a simulated environment based on Unity [3], and a physical environment based on RM-Testing. We evaluated the effectiveness of the three input validation approaches in keeping the autopilot program executing safely. The experimental results show that the selected input validation approaches are effective in guaranteeing the safe executions of context-aware systems, which improve the success rate of the robot car's executions by 82% in the simulated environment, and 50% in the physical environment. However, the effectiveness of these approaches does vary in different environment (32 percentage point). Thus, we believe that such works should be evaluated in the physical environment to better validate their effectiveness and usefulness.

In summary, this paper makes the following contributions:

- We built a testing platform, RM-Testing, for evaluating input validation approaches with a real context-aware system (DJI RoboMaster S1 robot car) in a physical environment.
- We implemented and adapted three up-to-date input validation approaches within our RM-Testing platform.
- We conducted extensive experiments in both the simulated environment and the physical environment to evaluate the effectiveness of the input validation approaches. The experimental results show the difference between the approaches' effectiveness in different environments.

The remainder of this paper is organized as follows. Section 2 introduces DJI RoboMaster S1 and three input validation methods associated with this work. Section 3 gives an overview of our testing platform for context-aware system. Section 4 introduces how we adapted the aforementioned approaches in our platform. Section 5 presents a primitive evaluation of the autopilot program based on our platform. Section 6 discusses related work, and finally Section 7 concludes this paper and discusses future work.

2 PRELIMINARIES

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

2.1 Self-driving car and DJI RoboMaster S1

Self-driving cars are usually equipped with many sensors to sense the environment. They analyze the current state based on the sensed environmental information, then use the predefined logic to determine their driving routes.

As a typical 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: 1) sensing the car's surrounding environment and receiving environmental information; 2) making decisions on the car's future route using the predefined 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 enables the robot car to move towards any direction and spin turn around within a small area. The robot is also equipped with a Wi-Fi module and an FPV camera, which enable one to connect the robot car to a computer and control its movement from a first-person-view (shorted as "FPV") in real time. However, as an educational robot, RoboMaster S1 has very limited sensors (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 [7]. Due to the limitation of physical measurement, error is inevitable as during the system's sensing phases, which could further affect the decision-making and action-performing phases, and finally leads the system into failure.
- **Uncertain scenarios.** To improve the productivity and cope with infinite kinds of environmental dynamics, software developers only hold certain assumption on a context-aware system's execution environment. Such simplification of environmental assumptions could the system facing uncertain scenarios that could cause the system's pre-defined logic fail[8].
- **Unfitted DL model input.** Deep learning (denoted as *DL*) models are trained with pre-collected data. However, a context-aware system could face complex and diverse running environments during its execution. If a DL model's running environment and training environment are completely different, the input 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[9].

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 effectiveness in both the simulated and the physical environment, with respect to improving the execution safety of context-aware systems. In the following part of this section, we briefly introduce the selected input validation techniques.

Constraints checking. Noise in sensing data usually causes context inconsistency. As a result, validating contexts helps preventing such inconsistency from being received by the system. One of the most popular approaches to detect context inconsistency is *constraints checking*[10][11][14]. Constraints describe the restrictions on the relationship between multiple pieces of contexts[13].

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

Figure 1: Constraint language syntax.

In this work, we study the effectiveness of Entire Constraints Checking (*ECC*)[11] approach in both the simulated and the physical environment. We use a constraint language based on first-order logic[11] to specify consistency constraints. The syntax of the constraint language is shown in Figure 1, where *bfunc* represents user-defined functions. The parameters of these functions are context instances and the return value is a boolean variable.

ECC checking context-consistency constraint in three steps as follows.

(1) Converting predefined consistency constraints into a consistency computation tree (CCT), where *bfunc* is represented as left node and other formulas are represented as nonleaf nodes.
(2) Specifying a post-order traversal of the CCT to calculate truth values of all nonleaf nodes. The truth value of the root node is the final checking result.
(3) Repairing the detect consistency error using three kinds of strategy, namely, drop-all, drop-latest and drop-random.

Invariant checking. Uncertain scenarios introduce unpredictable situation that beyond the capability of pre-defined logic of context-aware systems. Many systems use assertions to check whether the systems' surrounding environments enter 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 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, so the system can take corresponding repair measures to preventing failure.

Invariant detectors like Daikon[16] achieved great results in traditional software testing. Some researchers also proposed invariant generation templates for robotic systems[17]. In this work, we study the effectiveness of Context-aware Multi-Invariant Detection (*CoMID*) approach [19]. The main procedure of *CoMID* is as follows.

(1) Defining invariant templates according to actual needs. For example, $x >= C$ is an invariant template about the value range of variable x.
(2) Collecting several safe execution traces of the system. For template $x >= C$, we need to collect all values of variable x during the safe executions.
(3) Deriving an invariant which follows a predefined template and satisfies collected traces.
(4) Checking invariants while the subject context-aware system executing. If any of the invariant is violated, the system would perform a pre-defined remedy action to prevent the system from entering uncertain scenarios.

DL model input pruning. In recent years, deep learning approaches are widely used in many context-aware systems to assist their recognition of the physical environment (i.e., image recognition and speech recognition). Most of these approaches use supervised learning, whose DL models need to be trained using data collected from certain scenarios. As a result, the system's input from the running environment may be out of the model's capability.

In response to this problem, some DL model input validation methods for deep learning have been proposed. In this work, we study the effectiveness of *DISSECTOR* [9] approach in both the simulated and the physical environments. The main idea of this method is to distinguish and prune the inputs that exceeds the model's handling capability to prevent the data from being used in the actual decision-making. Since the remaining inputs are within its capability, they are more reliable to be used.

More specifically, *DISSECTOR* tracks how the model interprets its input and generates a *PVscore* to denote the input's validity. The

value range of *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.

3 A TESTING PLATFORM FOR CONTEXT-AWARE SYSTEM

We built a testing platform, *RM-Testing*, on DJI RoboMaster S1. The architecture of the platform is shown in Figure 2. Basically, the platform connects a DJI RoboMaster S1 robot car and a subject autopilot program under test. The platform mainly consists of four parts, namely *sense*, *control*, *input validation*, and *info*. The first two parts enables the subject program to sense the robot car's surrounding environment and control the robot car to move and rotate, respectively. The *input validation* implements the selected three input validation approaches, and provide high-quality environmental information to the subject program. The *info* collects execution information from all other modules for analyzing the program's execution states.

In the following parts of this section, we will describe the *sense*, *control* and info modules in detail. The *input validation* module will be introduced in the next section.

3.1 *Sense* in RM-Testing

The modules in *sense* enable the subject autopilot program to acquire environmental information. As discussed in Section II that the RoboMaster S1 only equipped with four contact sensors, we refitted the robot car to enrich its sensibility. We installed range sensors in the forward, rear, left, and right of the robot to obtain the horizontal distance between the robot and other objects. Since RoboMaster S1 does not open the underlying development board interface and wireless network module, we also installed 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 and waiting for response signal, then calculates the current reading based on the time difference between two signals. The ESP8266 WIFI module sends data to a computer in LAN.

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

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 camera, and feeds it to other modules that require the data. *Sensing input* connects providers (i.e., sensors and camera) and the consumers (i.e., *input validation* module and the autopilot program) of environmental information in a pub/sub manner. More specifically, both of the hardware that provide environmental data, and the modules that require environmental data first register their name and related data types in sensing input. Then *sensing input* would collect the raw data from the providers, and push the collected data to the consumers. *Sensing input* uses FIFO queues to store the collected environmental information, considering the different producing/consuming speed of the data.

3.2 *Control* in RM-Testing

The modules in *control* enables the autopilot program to control the robot car to move with enriched APIs, comparing with the original APIs provided by the RoboMaster S1 IDE. We implemented a *robot control* module based on the FPV controller of the robot car. With the original FPV controller, users can use keyboard to move and rotate the robot car. More specifically, pressing "W", "S", "A", "D", "left" and "right" keys would control the robot car move forward, back, left, right, rotate counterclockwise and clockwise, respectively. To enable the automatic control, we used pywin32 library to simulate keyboard actions of FPV mode, and encapsulated those simulated keyboard actions as methods for the autopilot program to invoke as shown in Figure 3.

3.3 *Info* in RM-Testing

To facilitate program debugging, error location, and data analysis, we implement a logging module based on the Python logging library. The execution record of any module in the platform will be output to the log files. We use three logging levels, including "DEBUG", "INFO", and "WARNING". The "DEBUG" information concerns the updates of corresponding variables in RM-Testing. The "INFO" information describes the pre-defined events produced from the RM-Testing platform. The "WARNING" information specifies the checking results produces by the implemented approaches in input validation module.

4 INPUT VALIDATING IN RM-TESTING

In this section, we describe the three input validation approaches implemented in RM-Testing platform. As we discussed in Section 1 that these existing approaches cannot be directly applied to real context-aware systems, we focus on the modifications we made during implementing these approaches.

4.1 Constraints checking

In this module, we implemented and modified the *ECC* approach for constraints checking with RoboMaster S1 robot car. We designed specific constraints that are effective for the robot-car-scenario based on previous work[10][11][14]. An effective constraint need to deal with errors that occur frequently in the scenario, so that they can efficiently improve the quality of validated context. In addition, the constraints should avoid missing detection (i.e., false negative instances) and false alarms (i.e., false positive instances). Thus, in order to design effective constraints, we first checked the execution traces of the robot car and analyzed main reasons for its failure. Based on the observation, we designed three types of constraint templates, which mainly concern the rapid changes of the range sensors' readings. After determining the constraint templates, we tried with different parameter settings and selected the best one according to their effectiveness.

The constraints we used in the system are shown in Figure 4. m and n are constants that 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 drop-latest strategy. Once

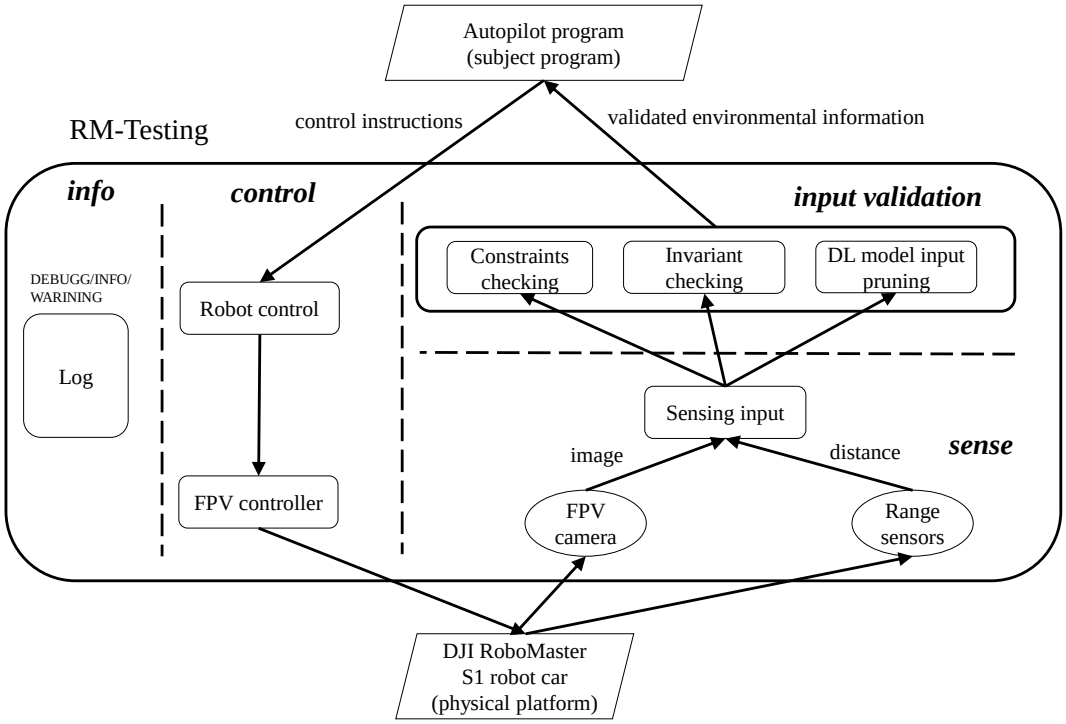


Figure 2: The architecture of RM-Testing platform.

(a)Before refitting. (b)After refitting. (c)FPV control.

Figure 3: Refitting and control of the car.

the context instance is validated, *constraints checking* stores it in a buffer for the autopilot program to use.

Since the speed of context-producing may be different from the speed of contraint checking, we use stacks to store the context instances pushed to the *contraint checking* module. Context of different patterns are pushed into different stacks. The stack's first-in-last-out property enables us to check constraints on the latest context instances, which would improve the responsiveness of the module.

4.2 Invariant checking

In this module, we implemented and modified the *CoMID* approach for invariant checking with RoboMaster S1 robot car. The main challenges to applying *CoMID* approach is to design effective invariant template. In the original *CoMID* approach, it simply uses Daikon invariant inference engine to derived the 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 in our RM-Testing platform, we cannot assume the availability of the subject program's source code. On the other hand, Daikon's invariant templates are designed for the internal variables of a program, while the environmental invariants mainly focus on the external variables of the environment.

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

Figure 5 presents the invariants templates we used. a and b are constants that can be specified automatically according to pre-collected execution traces. These invariants mainly focus on preventing the car from crashing into any obstacle.

We also designed 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.

4.3 Deep learning model input pruning

In this module, we implemented and modified the *DISSECTOR* [9] approach to validate the input of deep learning models. More specifically, we trained five sub-models of the origin image recognition

$$\forall \gamma_1 \in S_{forward}\ (\forall \gamma_2 \in S_{forward}($$
$$|\gamma_1.time - \gamma_2.time| < 1 \rightarrow |\gamma_1.distance - \gamma_2.distance| < m))$$

Constraint 1: The change of the forward sensor's reading shouldn't exceed *m* meters within 1 second.

$$\forall \gamma_1 \in S_{left}\ (\forall \gamma_2 \in S_{left}($$
$$|\gamma_1.time - \gamma_2.time| < 1 \rightarrow |\gamma_1.distance - \gamma_2.distance| < n))$$

Constraint 2: The change of the left sensor's reading shouldn't exceed *n* meters within 1 second.

$$\forall \gamma_1 \in S_{right}\ (\forall \gamma_2 \in S_{right}($$
$$|\gamma_1.time - \gamma_2.time| < 1 \rightarrow |\gamma_1.distance - \gamma_2.distance| < n))$$

Constraint 3: The change of the left sensor's reading shouldn't exceed *n* meters within 1 second.

Figure 4: Constraints used in evaluation.

$$d_{left} \geq a \ \wedge \ d_{right} \geq a$$

Invariant template 1: When the car is about to turning direction, it doesn't be too close to the obstacles on the left or right.

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

Invariant template 2: When the car is moving forward, the readings of the range sensors on the left or right don't change too rapidly.

Figure 5: Invariants used in evaluation.

model. Each sub-model is associated with specific layer of the origin model. The system will calculate a *PVsocre* for each input based on the origin model and five sub-models.

With the original *DISSECTOR* approach, 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 of an intersection, the autopilot program would fail to response to a road sign and move towards the correct direction. To prevent this situation, we designed 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 on one time and choose the image with the highest *PVscore* to be recognized by the autopilot program.

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.

RQ1: *Whether the selected approaches work effectively in preventing a context-aware system from failing?*

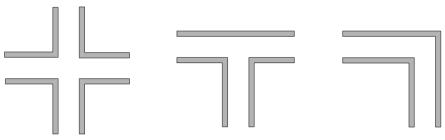


Figure 6: Predefined intersection types.

RQ2: *Whether the difference between a simulated environment and a physical environment affect the effectiveness of these selected approaches in preventing a context-aware system from failing?*

5.1 Evaluation Design

Scenario. The scenario we designed is a static urban road network, which consists of straight roads and intersections, simulating the real world. To simplify the scenario, we designed 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 predefined types (as shown in Figure 6), so any two roads are vertical or parallel. We also defined the starting point and ending point of

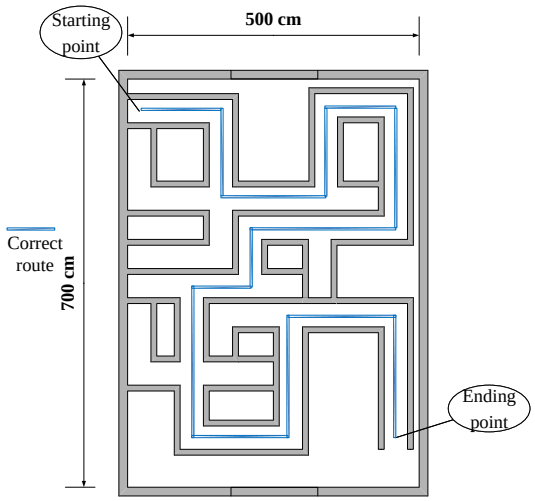


Figure 7: Design of the map used in evaluation.

the map. The choice of the starting and ending point guarantees the uniqueness of the correct route of the robot car. As a result, the robot car must turn the correct direction at each intersection in order to reach the ending point. We use road signs on the ground to indicate the correct direction in every interaction. The map we designed is shown in Figure 7

Task. In the initial state of the scenario, the robot car is placed at the starting point. Then the autopilot program controls the robot car toward the ending point. The program should adjust the car's actions in real-time based on the environmental data sensed by range sensors and camera. The task will fail if the car collides the fences on both sides of the road or turns in the wrong direction at any intersection during driving.

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. More specifically, 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 ending point.
- No wrong turning. The autopilot program should make sure that the car turns correct direction at all intersections. More specifically, the program uses a deep-learning-based image recognition module to identify the road signs and the correct direction at intersections.

The autopilot program's execution is considered "failed" when any of the two objectives are violated. If the autopilot program controls the robot car to reach the ending point with no failure, then this execution is considered as a success one. The quality of the program is measure by "success rate", which is the ratio of the number of success executions and the number of total executions.

Subject. The subject we used in the experiments was 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 reaching the ending point with a high success rate.

Input validation. When we put the subject autopilot program in the physical environment, the success rate for the car to reaching the ending point is lowered due to the fact that 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 used *constraints checking* to detect and repair the consistency error of sensor data, improving its reliability.
- We used *invariant checking* to detect and repair the car's abnormal state, avoiding collision as much as possible.
- We used *DL model input pruning* to improve the accuracy of image recognition.

5.2 Evaluation Setup

We constructed the scenario based on the aforementioned map in the physical and the simulated environments respectively (as shown in Figure 8). Configurations like object scale, car behavior, range sensor installation position and camera angle in these environments are exactly the same.

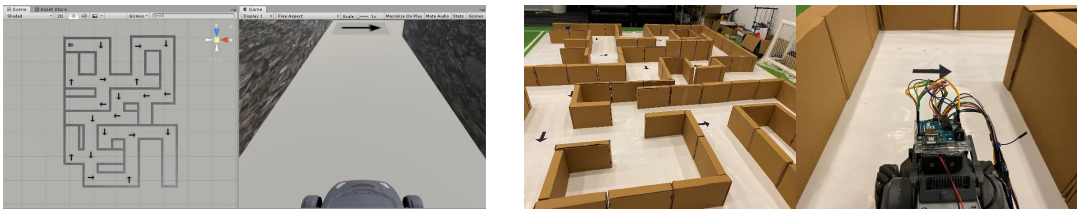
The physical Environment. We constructed the physical scenario on flat ground with a size of 500cm*700cm. We use white paper to cover the ground for two considerations. On the one hand, paper can make the road as flat as possible to prevent small potholes on the ground from affecting the car; on the other hand, the original color of the ground is inconsistent, and white paper helps to reduce impact on deep learning method. Then we used paper boxes as fences on both sides of the road. Finally, we posted road signs at all intersections in order to indicate the correct direction.

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

5.3 Evaluation Procedure

All the experiments are conducted on a laptop with an AMD Ryzen 7 4800U CPU @1.8GHz and 16GB RAM.

To answer research question RQ1, we compare the success rate of different configurations (whether methods are enabled or not). In the simulated environment, we conducted all groups of experiments as shown in Figure 1 and run 50 times with the same configuration for each group. In the physical environment, considering the experimental cost, we conducted experiments with configuration of group 1 and 8 (i.e. all methods are enabled or disabled) 50 times to simply observe the overall effectiveness of these three methods. These experiments are named *exp-1*. To better study the effectiveness of constraints checking and invariant checking from a more fine-grained perspective, we placed the car at the starting point with random displacement deviation and angle deviation, and observed whether it can pass the first intersection, named *exp-2*. We



(a) In simulated environment. (b) In physical environment.

Figure 8: Evaluation in the simulated and the physical environment.

Table 1: Configuration for each group.

Group	Configuration (enabled or not)		
	Constraints checking	Invariant checking	DL model input pruning
1			
2			✓
3		✓	
4		✓	✓
5	✓		
6	✓		✓
7	✓	✓	
8	✓	✓	✓

Table 2: Success rate of each group in the simulated environment.

Group	Number of success	Success rate
1	0	0%
2	0	0%
3	14	28%
4	7	14%
5	0	0%
6	1	2%
7	36	72%
8	41	82%

ran *exp-2* with configuration of all groups shown in Figure 1 and 50 times for each group, too.

To answer research question RQ2, we compare the success rate of the simulated environment and the physical environment in *exp-1*.

In *exp-1*, we record the success rate, the number of constraints checking and invariant checking and the prediction result of every image. In *exp-2*, we record the success rate.

5.4 Evaluation Results and Analyses

RQ1 (the effectiveness of input validation)

We will firstly discuss experiments in the simulated environment.

Table 2 gives an overview of the results of *exp-1* on the success rate by the eight groups under comparison. When all three methods are disabled, the car never reaches the ending point. We think the main reason leading to this phenomenon is that the ending point is far away from the starting point, so the accumulated deviation causes the car to deviate from correct route. With all three methods enabled, the success rate increases to 82%, indicating that these methods can effectively prevent the car from failing.

To observe the effectiveness of each method, we also calculate success rate from the perspective of each method, as shown in Table 3. We observe that constraints checking increases the success rate by 28.5% and invariant checking increases the success rate by 48.5%, which shows these two methods' effectiveness. However, input validation decreases the success rate by 2.0%. We think the main reason is that the accuracy of the original model is already at a high level (about 94.8%), so failure caused by prediction errors is rare compared with random errors or mechanical deviations. Therefore, this metric cannot show the effectiveness of input validation. On the other hand, decreasing by 2.0% is within reasonable error range.

We further investigate the effectiveness of input validation for deep learning models by analyzing accuracy. For all images in experimental group with input validation method enabled, we predict their labels with the original deep learning model (input validation method is disabled now), then observe the change of accuracy. As shown in Table 4, input validation method helps increase the accuracy by 1.6%. Since the base was high, we think it has been a big improvement.

The results of *exp-2* is shown in Table 5. We can find that constraints checking helps increase the success rate of *exp-2* by 9.0% and invariant checking help increase by 26.0%.

Then we study the effectiveness of input validation in the physical environment. Unfortunately, even with all three methods enabled, the robot car failed to reach the ending 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 comparing with it 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 system's accuracy in sign-recognizing reaches 90%, the probability for it to control the robot car to reach the ending point is barely over 20%, which could be father lowered by other uncertain factors.

As a result, we measure the proportions of the robot car to reaching the fifth and the sixth intersections, instead of the successful rate of it to reaching the ending point. The results are shown in Table 6. With all three methods enabled, the proportions of reaching the 5th and 6th intersections increases by 50% and 24%, respectively,

Table 3: Success rate grouping by method in the simulated environment.

Method	Number of rounds	Number of success			Success rate		
		Disabled	Enabled	Change	Disabled	Enabled	Change
Constraints checking	200	21	78	57	10.5%	39.0%	28.5%
Invariant checking	200	1	98	97	0.5%	49.0%	48.5%
DL model input pruning	200	53	49	-4	26.5%	24.5%	-2.0%

Table 4: Accuracy change in the simulated environment.

Group	Number of samples	Accuracy		
		Disabled	Enabled	Change
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	1569	94.8%	96.4%	**1.6%**

Table 5: Success rate change of *exp-2* in the simulated environment.

Method	Number of rounds	Success rate		
		Disabled	Enabled	Change
Constraints checking	200	74.0%	83.0%	9.0%
Invariant checking	200	65.5%	91.5%	26.0%

Table 6: Proportions of reaching different intersections in the physical environment.

Group	% of reaching the 5th intersection	% of reaching the 6th intersection
1 (all disabled)	4%	0%
8 (all enabled)	54%	24%

indicating these methods can also effectively prevent the car from failing in the physical environment.

Therefore, we answer **research question RQ1** as follows.

The selected approaches work effectively in preventing a context-aware system from failing. In general, these methods can increase the success rate of the car arriving the ending point by 82% in the simulated environment, and increase the proportion of reaching the 5th intersection by 50% in the physical environment. From the perspective of each approach, constraints checking increases the success rate of the car passing the first intersection by 9.0% and invariant checking increases it by 26.0%. Input validation increases the accuracy of image classification by 1.6%.

RQ2 (the compare of input validation's effectiveness in the physical environment and the simulated environment)

As mentioned above, the proportion of the car passing all 15 intersections and reaching the ending point reaches 82% in the simulated environment, while the proportion of it reaching the 6th intersection is 24% in the physical environment. Although these methods are effective in both environments, their effectiveness in the physical environment is not as significant as that in the simulated environment.

Therefore, we answer **research question RQ2** as follows.

*The difference between the simulated environment and the physical environment **does** affect the effectiveness of these selected approaches in preventing a context-aware system from failing. The proportion of the car passing all 15 intersections reaches 82% in the simulated environment while the proportion of it reaching the 6th intersection is 24% in the physical environment. The approaches' effectiveness in the physical environment is not as significant as that in the simulated environment.*

5.5 Threats to validation

The major threat to the validation of the result of our evaluation is the difference between the configurations of the simulated and the physical environment. As we discussed above, the reason that we use a smaller configuration in the physical environment is the autopilot program's failing in leading the robot car to the ending point. Nevertheless, we believe our current experiments could already indicates the different performance of input validation in the simulated environment and in the physical environment. What is more, the different configuration itself also validates the differences between a context-aware system's execution in a simulated environment and in a physical environment. This echoes our argument in Section 1 that CASs-based works should be evaluated in the physical environment.

Another major threat to the validity of our evaluation is the selection of the subject. We only use the platform for conducting experiments. This may harness the generalization of our conclusions. Conducting experiments in both the simulated environment and the physical environment requires full understanding of the testing platform, which restricts our choice of potential subjects. Nevertheless, we believe that RM-Testing is a good evaluation subject since it represents and integrates various kinds of self-driving-related context-aware services, including collision avoidance, disengagement detection, and road sign recognition.

6 RELATED WORKS

Constraint checking. Nentwich et al. [20] propose xlinkit for repairing inconsistent XML documents. This framework builds on an incremental checking model. Egyed et al. [12] focus on repairing inconsistency error in UML models. These pieces of works were evaluated from using pre-collected or generated context data.

Xu et al. [11] propose 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* approach were evaluated in the simulated environments using real context data.

Invariant checking. Invariant checking enables program to detect potential abnormal state at runtime. Xu et al. [15] propose monitoring runtime errors for an application and relating them to responsible defects in the application. Qin et al. [19] explore multi-invariant detection based on context-based trace grouping. These two pieces of works are evaluated in both the simulated and the physical environments.

Besides, Ramirez et al. [21] propose discovering combinations of environmental conditions to trigger specification-violating behaviors. Aliabadi et al. [18] explore mining dynamic system properties around time. Wang et al. [26] propose identifying program points where the system's behavior may be affected by context changes. These pieces works are evaluated in the simulated environments using generated context data.

Deep learning model input pruning. Input validation for deep learning model could greatly improve the performance of DL models in terms of their accuracy in predicting. Chu et al. [23] focus on data cleaning for more qualified training data. Pei et al. [22] converted the corner-case generation problem to joint optimization problem. Tian et al. [27] propose a testing tool for detecting abnormal state of DNN-driven vehicles that can potentially lead to crashing. Wang et al. [9] track each input's interpretation for estimating its validity. All these pieces of works evaluated their approaches using static datasets.

As discussed above, most related works simply evaluated the proposed approaches in a simulated environment, and only few of them conducted experiments in a real physical environments.

7 CONCLUSION AND FUTURE WORK

In this paper, we built a testing platform, RM-Testing, based on DJI RoboMaster S1 and tested constraints checking, invariant checking, and deep model input pruning in both the simulated environment and the physical environments. Our experimental evaluation validates the three methods' effectiveness in preventing a context-aware system from failing in both environments. It also indicates that evaluating approaches in only a simulated environment is not convincing considering the difference between the simulated environment and the physical environment. Thus, we believe that *CASs-based works should be evaluated in the physical environment* in order to demonstrate their effectiveness and usefulness.

Our work still has room for improvement. For example, our current experiments reveal the differences of the simulated environment and the physical environment. Due to the limitation of effort and space, we failed to figure out the root cause of this discrepancy. We will further recognize the factors that contribute the discrepancy, such as sensor noise and mechanical deviation, and perform an abrasion study to investigate the impact of those factors.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation (Grants No 61932021, 61902173) of China and Natural Science Foundation of Jiangsu Province (Grants No BK20190299). The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] Lü J, Ma X, Huang Y, et al. Internetware: a shift of software paradigm[C]//Proceedings of the First Asia-Pacific Symposium on Internetware. 2009: 1-9.
[2] https://www.dji.com/cn/robomaster-s1.
[3] https://unity.com.
[4] https://link.springer.com/chapter/10.1007/978-3-030-00262-6_11.
[5] https://www.bbc.com/news/technology-48308852.
[6] National Transportation Safety Committee of Indonesian, "Preliminary Report on the Lion Air B737 MAX 8 accident,", https://reports.aviation-safety.net/2018/20181029-0_B38M_PK-LQP_PRELIMINARY.pdf.
[7] Xi W, Xu C, Yang W, et al. How context inconsistency and its resolution impact context-aware applications[J]. Journal of Frontiers of Computer Science and Technology, 2014, 8(4): 427.
[8] Esfahani N, Malek S. Uncertainty in self-adaptive software systems[M]//Software Engineering for Self-Adaptive Systems II. Springer, Berlin, Heidelberg, 2013: 214-238.
[9] Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. DISSECTOR: Input Validation for Deep Learning Applications by Crossing-layer Dissection. In Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE 2020), pp. 727-738, Seoul, South Korea, May 2020.
[10] Nentwich C, Emmerich W, Finkelsteiin A, et al. Flexible consistency checking[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2003, 12(1): 28-63.
[11] Xu C, Cheung S C, Chan W K, et al. Partial constraint checking for context consistency in pervasive computing[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2010, 19(3): 1-61.
[12] Egyed A. Fixing inconsistencies in UML design models[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 292-301.
[13] Gehrke J, Madden S. Query processing in sensor networks[J]. IEEE Pervasive computing, 2004, 3(1): 46-55.
[14] Tarr P, Clarke L A. Consistency management for complex applications[C]//Proceedings of the 20th international conference on Software engineering. IEEE, 1998: 230-239.
[15] Xu C, Cheung S C, Ma X, et al. Adam: Identifying defects in context-aware adaptation[J]. Journal of Systems and Software, 2012, 85(12): 2812-2828.
[16] Ernst M D, Perkins J H, Guo P J, et al. The Daikon system for dynamic detection of likely invariants[J]. Science of computer programming, 2007, 69(1-3): 35-45.
[17] Jiang H. Invariant Inferring and Monitoring in Robotic Systems[J]. 2014.
[18] Aliabadi, M. R., Kamath, A. A., Gascon-Samson, J. and Pattabiraman, K., "ARTINALI: dynamic invariant detection for cyber-physical system security," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 349-361.
[19] Y. Qin, T. Xie, C. Xu, A. Astoga and J. Lu, "CoMID: context-based multi-invariant detection for monitoring cyber-physical software," in *IEEE Transactions on Reliability*, to be published, 2019.
[20] Nentwich C, Capra L, Emmerich W, et al. xlinkit: A consistency checking and smart link generation service[J]. ACM Transactions on Internet Technology (TOIT), 2002, 2(2): 151-185.
[21] Ramirez A J, Jensen A C, Cheng B H C, et al. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems[C]//2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011: 568-571.
[22] Pei K, Cao Y, Yang J, et al. Deepxplore: Automated whitebox testing of deep learning systems[C]//proceedings of the 26th Symposium on Operating Systems Principles. 2017: 1-18.
[23] Chu X, Morcos J, Ilyas I F, et al. Katara: A data cleaning system powered by knowledge bases and crowdsourcing[C]//Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015: 1247-1261.
[24] Kulkarni D, Tripathi A. A framework for programming robust context-aware applications[J]. IEEE Transactions on Software Engineering, 2010, 36(2): 184-197.
[25] Sama M, Elbaum S, Raimondi F, et al. Context-aware adaptive applications: Fault patterns and their automated identification[J]. IEEE Transactions on Software Engineering, 2010, 36(5): 644-661.
[26] Wang Z, Elbaum S, Rosenblum D S. Automated generation of context-aware tests[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 406-415.
[27] Tian Y, Pei K, Jana S, et al. Deeptest: Automated testing of deep-neural-network-driven autonomous cars[C]//Proceedings of the 40th international conference on software engineering. 2018: 303-314.