DESIGN AND EVALUATION OF TEST ORACLES FOR CONTROL-CPS AND REINFORCEMENT LEARNING SOFTWARE

SHIYU ZHANG

PhD

The Hong Kong Polytechnic University

2026

The Hong Kong Polytechnic University
Department of Computing

Design and Evaluation of Test Oracles for Control-CPS and Reinforcement Learning Software

Shiyu Zhang

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
July 2025

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

Signature: _____

Name of Student: Shiyu Zhang

Abstract

In this thesis, we focus on the challenge of designing test oracles for control cyber-physical systems (control-CPSs), aka the oracle problem for control-CPSs. This challenge arises because, for conventional control-CPSs, a given input (high level control goal) can correspond to multiple legitimate outputs (specific physical trajectories of the controlled plant). This problem is worsened for control-CPSs using reinforcement learning (RL) to generate controllers: not only the legitimate physical trajectories are not unique, the legitimate controllers generated by the RL are not unique either. To address this challenge, we design and evaluate three novel test oracles. For general control-CPS software, we propose a Transformer-based oracle (TO) and compare it with the known best practice of Auto-Regressive System Identification (AR-SI) oracle (AO), focusing on software fault localization (SFL) quality metrics. Our comparison results show that TO and AO perform similarly in SFL quality, but TO has a slightly better performance than AO in terms of latency; AO has a lower false positive rate and TO has a lower false negative rate. As for overall performance, there is no significant differences between TO and AO. For RL software, we propose two test oracles: a fuzzy logic-guided oracle and a Lyapunov stability theory-based oracle (LPEA). We compare both oracles with the mainstream best practice of human oracle (HO). Evaluation results show that the fuzzy logic-guided oracle is not significantly better than the human oracle, except for a few metrics. However, the LPEA oracle can significantly outperform the human oracle in most of the metrics. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%,

34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

Publications Arising from the Thesis

1. Shiyu Zhang, Haoyang Song, Qixin Wang, Henghua Shen and Yu Pei, "A Test Oracle for Reinforcement Learning Software based on Lyapunov Stability Control Theory", in *IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 502-513, 2025. Received ICSE 2025 Distinguished Paper Award.

2. Shiyu Zhang, Haoyang Song, Qixin Wang and Yu Pei, "Fuzzy Logic Guided Reward Function Variation: An Oracle for Testing Reinforcement Learning Programs", in *arXiv preprint*, arXiv:2406.19812, 2024.

3. Shiyu Zhang, Wenxia Liu, Qixin Wang, Lei Bu and Yu Pei, "A Comparison of Transformer and AR-SI Oracle for Control-CPS Software Fault Localization", in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 95-100, 2023.

O Muses, let inspiration dwell within us forever.

Acknowledgments

First of all, I thank my chief supervisor, Prof. Qixin Wang. He is a dedicated computer engineer/scientist with profound expertise in his field. In our academic discussions, he explains technical details effortlessly, and shows genuine curiosity when exploring new topics. His passion for engineering/science and truth inspires me and shows what true scholarship means. Prof. Wang also has a wide-range of interests, including painting, music, ancient poetry, and philosophy. Particularly, he taught me not only technical details on how to do research, but also how to see things at a philosophical level.

Secondly, I thank my co-supervisors: Prof. Asif Usmani and Prof. Linda Fu Xiao. Both professors provided invaluable guidance and support throughout the Surefire project. My involvement in this project significantly deepened my expertise on urban smart firefighting systems. It also allowed me to establish connections with numerous outstanding scholars and peers. Notably, through this project, I was honored to deliver an academic demonstration at Tsinghua University, an experience for which I am deeply grateful.

Thirdly, my deepest gratitude goes to all my friends and family for their unwavering support. You provided the emotional anchor throughout my studies, granting me strength to face all kinds of challenges on my academic journey.

Last but not least, I thank reviewers for their comments that improve my papers.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber-Physical Systems (CPS) [56, 77, 80] are systems that integrate cyber computing with applications in the physical world. In this thesis, we focus on CPSs related to control applications, aka *control-CPSs* [13].

Many control-CPSs are mission/safety critical, e.g. avionics, autonomous driving [91], robotic arms for manufacturing or surgeries [63]. For such control-CPSs, dependability is essential [37]. In pursuit of the control-CPSs' dependability, one important aspect is to pursue the correctness of the control-CPSs' software. To pursue the correctness of the control-CPS software, testing and debugging are inevitable. As the control-CPS software scales up (e.g., a consumer level quadcopter's software can easily exceed millions of lines of source code [1]), testing and debugging automation are inevitable. To automate software testing and debugging, we need *test oracles* (simplified as "*oracles*" in the following): a program that can automatically label the correctness of a piece of software/system by just looking at the software/system's input and output.

Unfortunately, how to design oracles is highly application dependent: a generic routine is lacking. Oracle design hence can be extremely difficult. This is the well-known *oracle problem* [15]. For control-CPSs, the oracle problem can be even bigger. Unlike

conventional software, where the correct output for a given input is usually unique, for control-CPSs, given an input, there can be many correct outputs. For example, to fly an airplane (a typical control-CPS) from Hong Kong to New York (i.e. given the input: the source and destination cities), there can be infinite many acceptable ways to fly.

In this thesis, we shall propose/evaluate solutions for the control-CPS oracle problem.

Particularly, Chapter 2 explores test oracles for control-CPS, with a comparison between the traditional *Auto-Regressive System Identification (AR-SI) Oracle* and a novel *Transformer-based Oracle*. Leveraging the strengths of transformer models in time-series prediction, this work evaluates the two approaches in localizing faults in control-CPS software, particularly through trajectory data analysis. The content of Chapter 2 is published in the following IEEE paper:

-

Chapter 3 and 4 further focus on addressing the oracle problem for *Reinforcement*

Learning (RL) [87]. This is important because RL has great potentials in generating AI models as controllers for complex control-CPSs, especially when the control-CPS is so complex that it can hardly be analyzed with conventional methods [21] [59]. In this sense, RL software is a part of the control-CPS software. Meanwhile, testing/debugging RL software pose new challenges to the control-CPS oracle problem: now not only the legitimate controller behaviors are not unique, the legitimate controllers generated by the RL are not unique either. It is hard to label which controller is "correct"/"incorrect". All that we can measure (as measuring AI models generated by machine learning) is the controllers' performances against a test set. However, the performance metrics are usually multidimensional, statistical, and dependent on the test set. It is hard to define which performances are better/worse, not to speak of labeling performances as "correct"/"incorrect".

With the above said, Chapter 3 proposes *fuzzy logic based oracle* for testing/debugging RL software. This oracle generates I *intended-policies*, each covers only a few reference states with the corresponding reference actions. Fuzzy logic is then used to generate the reward function to cover the entire state space. Then the training stage starts. For each intended-policy and reward function, E epochs of training are carried out. Fuzzy logic is used again to evaluate the compliance of the epoch's learning with the intended-policy. The E training epochs hence result in an *epoch-compliance-value time-series* (note the epoch-compliance-value is different from RL training loss: i) only one epoch-compliance-value is generated per training epoch, but n loss values are generated in a training epoch involving n back-propagation steps; ii) epoch-compliance-value is based on the *ground truth* intended-policy, while the loss is based on the RL program's understanding/estimation of the reward function). If the epoch-compliance-value time-series shows an improving trend, then the RL learning is considered normal; otherwise, the RL learning is considered abnormal. If for the I intended-policies, majorities of the RL learning are normal, then we consider the RL program bug-less; otherwise, buggy. We compared the performance of the proposed

fuzzy logic based oracle with human oracles, and the results show that the fuzzy logic based oracle still lags behind the human oracle on many performance metrics. However, this work reveals the difficulty of the oracle problem for RL software, and inspired us to further exploit domain knowledge to design better oracles. This leads us to the success of the work of Chapter 4. The content of Chapter 3 is archived as follows:

- Shiyu Zhang, Haoyang Song, Qixin Wang and Yu Pei, "Fuzzy Logic Guided Reward Function Variation: An Oracle for Testing Reinforcement Learning Programs," in arxiv. DOI: 10.48550/arXiv.2406.19812

Chapter 4 proposes a *Lyapunov stability control theory based oracle* for testing/debugging RL software. This oracle exploits the Lyapunov stability control theory, specifically, the Lyapunov potential energy concept and the Lyapunov potential energy decreasing behavioral rule, to construct the reward function in RL training. Our conjecture is that if the RL software is correct, it should figure out the hidden Lyapunov potential energy concept and the Lyapunov potential energy decreasing behavioral rule. Otherwise, the RL software is considered incorrect. We also compared the performance of the proposed oracle with the human oracle. The results show that our proposed oracle can significantly outperform the human oracle. The content of Chapter 4 is published in the following paper, which won a Distinguished Paper Award from the IEEE/ACM International Conference on Software Engineering (ICSE):

- Creative Commons License, Shiyu Zhang, Haoyang Song, Qixin Wang, Henghua Shen and Yu Pei, "A Test Oracle for Reinforcement Learning Software based on Lyapunov Stability Control Theory," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025)*, pp. 502-513. DOI: 10.1109/ICSE55347.2025.00074.

All the research reported in this thesis was performed under the oversight of the Hong Kong Polytechnic University IRB, and consent from the human subjects in the research was obtained.

Chapter 2

A Comparison of transformer and AR-SI Oracle For Control-CPS Software Fault Localization

2.1 Introduction

Control-CPS refers to those software systems that interact with the physical world [81]. With the development of Industry 4.0, control-CPS is increasingly common in our life [39]. Typical control-CPSs include smart grids, autonomous automobile systems, medical monitoring, industrial control systems, robotics systems, etc [40]. The wide application of these control-CPS in everyday life may lead to some safety concerns. For example, an out-of-control autonomous automobile is likely to pose a safety threat to pedestrians on the road. Therefore, developers should conduct comprehensive debugging for these systems to ensure there are no safety or other severe problems [52].

As nowadays control-CPSs reaching millions of lines of source code, traditional human-flesh debugging is no longer sufficient. We need automated debugging tools. One

major category of such tools is those for *software fault localization* (SFL), i.e. the tools to locate suspicious buggy lines in the source code. The SFL tools can be further categorized into various families [99]. Two of the main stream families are the program spectrum analysis SFL [10] and statistical analysis SFL [58]. Both families use the big data idea. A typical working flow of SFL and oracle is in Fig. 2.1, quoted from [36]. First we generate a large number (i.e. big data) of code traces (files contain blocks that program pass through; The blocks are usually in form of line numbers or function names, etc). Then, we examine these large number of code traces: if a program block often appears in the buggy code traces, but rarely appears in the correct code traces, then the program block is suspected to be buggy. A natural question then is, how to determine a code trace is buggy or correct. The tools to make such decisions is called *oracles* [15].

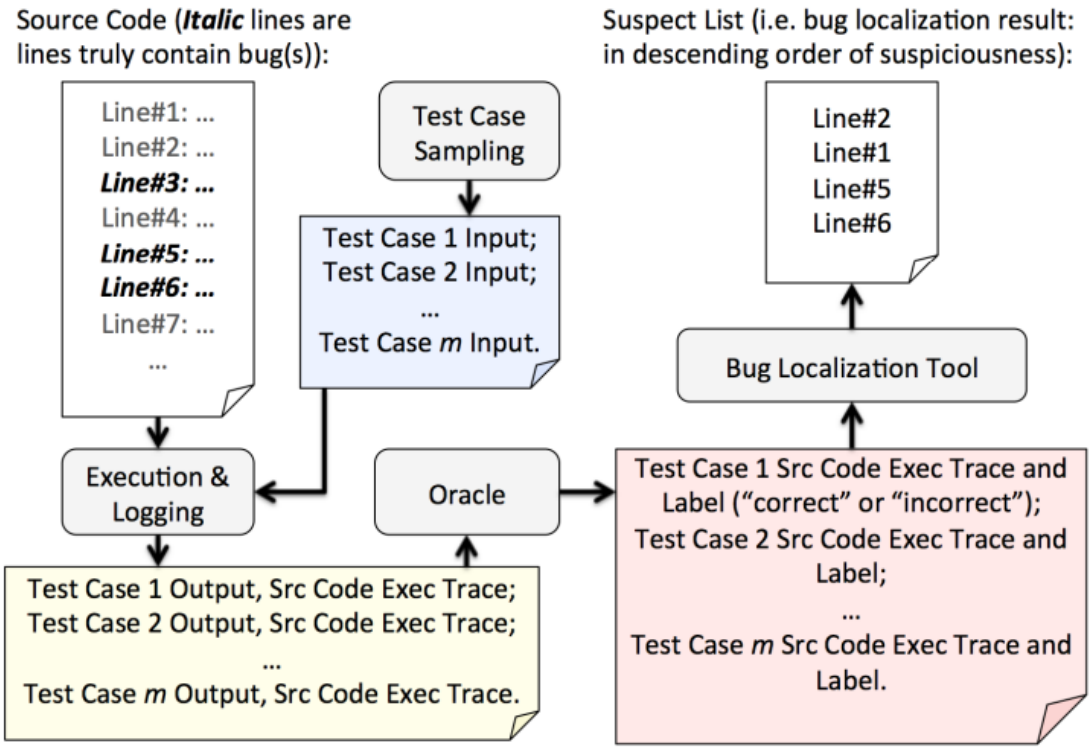


Figure 2.1: Typical work flow of (program spectrum, statistics, etc) bug localization (aka software fault localization, simplified as "SFL") tools (quoted from [36])

Oracle design difficulty is heavily application dependent. For some applications (such as sorting), oracles are very easy to design. But for most other applications, oracles are extremely difficult to design (this is the well known *oracle problem* [15]). Control-CPS oracle designs, unfortunately, usually belong to the latter. For example, for a control-CPS to fly a drone from point A to point B, the output of the control-CPS is the trajectory of the drone (see Fig. 2.2). The oracle needs to judge merely from this trajectory, the correctness of the control-CPS execution (note there could be many ways to fly from A to B, and not every buggy execution leads to drone crash).

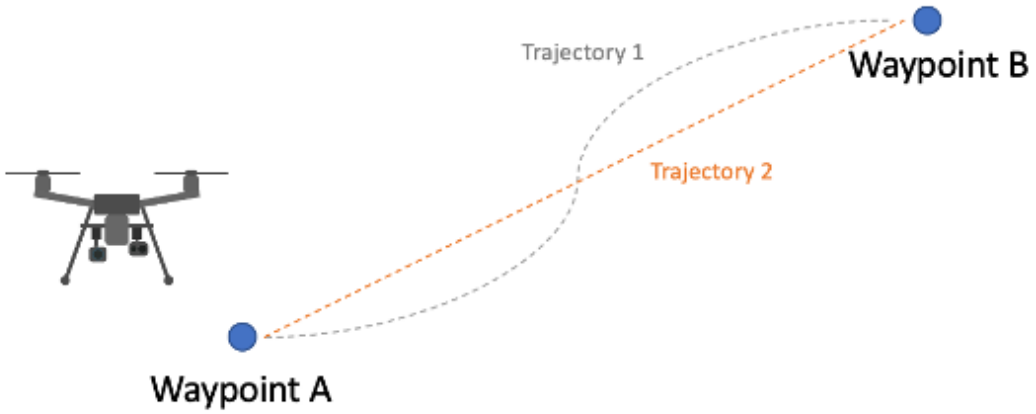


Figure 2.2: Oracle of control-CPS output

So far, control-CPS oracle design is still mostly an open problem space. To our best knowledge, *auto-regressive system-identification* (AR-SI) oracle [35] is the most widely adopted control-CPS oracle (another alternative is to use human to label the correctness of control-CPS code traces but this is infeasible in the context of big data based automated SFL).

The AR-SI oracle believes that a buggy control-CPS will generate jitters in the outputted physical-subsystem trajectories (simplified as "*trajectories*" in the following). The method uses AR-SI to identify the math model of the control-CPS. Then, it uses the identified model to predict the future trajectory. If the difference between

the actual trajectory and the predicted trajectory is smaller than a preset threshold, then the trajectory (and the corresponding code trace) is marked correct; otherwise, marked buggy.

AR-SI plays the function of time series prediction in the oracle approach mentioned above. Time series prediction, however, is also a focus application of AI. Recently, transformer [93] emerges as a game changer in the domain of AI based time series prediction [86]. Naturally, people wonder if transformers can be used to build control-CPS oracles, and how well they can perform compared to the AR-SI control-CPS oracles.

This paper aims to make an initial attempt to answer the above concerns.

We propose a transformer control-CPS oracle (simplified as TO in the following), and compare it with the AR-SI control-CPS oracle (simplified as AO in the following). Specifically, our TO assumes the existence of an earlier well-debugged version of the control-CPS. We use this well-debugged version to train a transformer based trajectory time series predictor. We use this predictor to predict the trajectories of a new version of the control-CPS (the to-be-debugged version). If the predicted trajectory is significantly different from the actual trajectory, the corresponding actual trajectory (and the corresponding code trace) is labeled buggy; otherwise, correct.

We tested the TO, and compared its performance with the AO. The results show that: i) in terms of SFL accuracy and latency, the TO and the AO are similar; ii) in terms of false positive rate, the AO performs significantly better; and iii) in terms of false negative rate, the TO performs significantly better.

The rest of the paper is structured as follows: Section 2.2 specifies the control-CPS context; Section 2.3 presents our TO solution; Section 2.4 compares TO with AO; Section 2.5 discusses related work; and Section 2.6 concludes the paper.

2.2 Problem Context

2.2.1 Control-CPS Model

In this paper, we focus on a control-CPS as shown in Fig. 2.3. We can instrument the source code in the cyber-subsystem of the control-CPS (specifically, add logging instructions to record the code traces), but we do not understand the source code, at least not before the automated SFL gives us a list of suspected buggy source code blocks.

Users send operation instructions to control-CPS. Receiving the operation instructions, the cyber-subsystem senses and actuates the physical-subsystem (or the simulator of the physical-subsystem). During operation, users are able to get the status information of control-CPS.

We regard the operation instructions given by the user as system input $U(t)$, $t \in [0, +\infty)$. User operation instructions are recorded before sending to the control-CPS, with a time interval of T. We denote U_h as the hth sampled $U(t)$. $U(t)$ stays unchanged during $[hT, (h+1)T)$. In other word, user send an operation instruction to control-CPS every time interval T.

$Y(t)$ is the physical trajectory generated by the system. We sample it periodically, and use Y_i to represent the ith sampled $Y(t)$. The time interval is Δ.

2.2.2 Assumptions

Assumption 1: With output $Y(t)$ frequency domain upper bound $F_{max} < \infty$(Hz), $Y(t)$ can be considered as continuous and differentiable. Corresponding to Assumption 1, control-CPS design has the following empirical rule on how to set Δ [35].

Assumption 2: The cyber system has to sample quickly enough ($\Delta \leq 1/(20F_{max})$)

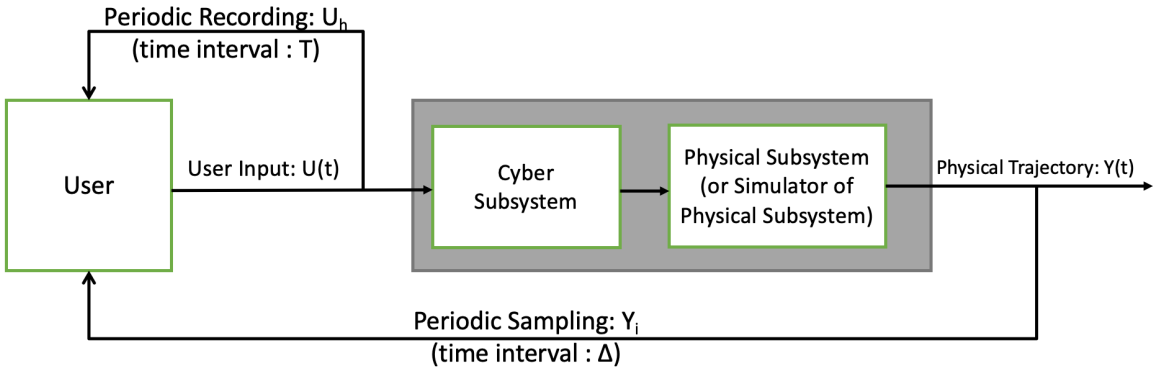


Figure 2.3: Control-CPS Model

for the physical system to treat it like a continuous signal processor.

To repeat the experiment, we created a software (named "monkey") that would automatically send commands to the control-CPS to move to the target position every T time period. The "monkey" will also log the trajectory of the control-CPS at a time t while operating. To make the experiment easy to repeat, we make the control-CPS running in simulator. After each execution, related code trace and associated physical trajectory will be recorded. We need to oracle the correctness of the physical trajectory. Labeling the correctness of physical trajectories leads to the control-CPS oracle problem discussed in Section I.

2.3 Solution

2.3.1 Heuristics

A typical control-CPS model is Fig. 2.3. The input is U_h and output is Y_i. We can apply a sliding window with the size of p to Y_i since they are dispersed trajectories dots sampled from continuous trajectory $Y(t)$. As illustrated in Section 2.2, $Y(t)$ represents trajectory and $U(t)$ represents user input. In our experiment, $U(t)$ is the target state (aka *reference state*) that users want the physical-subsystem to reach at

time t. $U(t)$ hence is also called the *reference trajectory*.

For any $Y(t)$, there must be a relevant $U(t)$. By concatenating $Y(t)$ and $U(t)$, we will get $Y'(t) \stackrel{\text{def}}{=} (Y(t), U(t))$. Suppose the window size is p, then we can define a small part of trajectory containing waypoint information:

$$T_q \stackrel{def}{=} \{Y'_{q-p+1}, Y'_{q-p+2}, ..., Y'_q\}, Y' \in \mathbb{R}^d, \tag{2.1}$$

T_q here represents the qth sliding window of trajectory (together with waypoint information). A transformer $f_\theta : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$ "transforms" a collection of n objects in $\mathbb{R}^d$ to another collection of n objects in $\mathbb{R}^d$ [89]. We built a transformer to predict the next sliding window of trajectory using the previous one

$$\hat{T}_{q+1} = f_\theta(T_q) \tag{2.2}$$

$$\hat{T}_{q+1} \stackrel{def}{=} \{\hat{Y}'_{q-p+2}, \hat{Y}'_{q-p+3}, ..., \hat{Y}'_{q+1}\}, Y' \in \mathbb{R}^d, \tag{2.3}$$

In (2.3), $\hat{T}_{q+1}$ represents the predicted sliding window, $\hat{Y}'$ represents the predicted items in the sliding window. $\hat{Y}_{q+1}$ represents the predicted trajectory combing with the relevant waypoint. Taking our the first few dimensions of $\hat{Y}'$, we will get Y_{q+1}, which represents the predicted trajectory point. When Y_{q+1} is available, we know the

$$transformer\ prediction\ error : e_{q+1} \stackrel{def}{=} \hat{Y}_{q+1} - Y_{q+1} \tag{2.4}$$

Heuristics 1: F_{max}^{actual} stands for the actual maximal frequency component of the continuous output of control-CPS. Normally, a control-CPS's output should have $F_{max}^{actual} \leq \frac{F_{max}}{10}$, or equivalently $\frac{10}{F_{max}} \leq \frac{1}{F_{max}^{actual}}$ [35]. For instance, despite the fact that an engine may shake at a maximum frequency of 100Hz, a typical design would not permit such high shaking (under controlled sounds). The design must keep the shaking to under 10Hz.

Heuristics 2: When $F_{max}^{actual} < \infty(HZ)$, $Y(t)$ can be considered as linearizable in small enough sliding time window ($p\Delta \leq 1/(20F_{max}^{actual})(sec)$) [31] [99]. After training, the transformer model will predict a reasonable state value based on the training set. the transformer prediction error magnitude (when $p\Delta \leq 1/(20F_{max}^{actual})$) should be small; a magnitude outlier suggests that something unusual—and hence probably buggy is taking place. The transformer prediction error can serve as an oracle in this regard.

Now let us discuss the setting of p. With **Heuristics 1**, **Heuristics 2**, and **Assumption 2**, we have if $p = 10$, then $p\Delta \leq 10/(20F_{max}) \leq 1/(20F_{max}^{actual})$. In this way, the prerequisite for **Heuristics 2** holds. Hence, by setting p to 10, we can apply **Heuristics 2** to the oracle for control-CPS SFL.

2.3.2 Transformer Oracle: Solution

Based on the heuristics of Section III-A, we propose the oracle and code trace preparation methodology (following the overall design of the AR-SI oracle approach proposed in [35]). Our new approach is called the "*transformer oracle*", as shown in Fig. 2.4.

Another classic automatic oracle for control-CPS is the "*AR-SI oracle*" [35]. The only difference is replacing transformer in **Step1-Task2** with AR-SI, as shown in Fig. 2.5.

2.4 Evaluation

Ardupilot is a very popular test-bed in the CPS field. It has more than 300,000 lines of code and is a very complicated open-source CPS [11]. Applicable devices for Ardupilot include multirotor drones, helicopters, rovers, boats, etc. ArduCopter is the multirotor drone part of Ardupilot. In this section, we evaluate the proposed transformer oracle (TO) and up-to-date AR-SI oracle (AO) on ArduCopter with three

Step1

Task1: Simulate the control-CPS using the simulation platform of Fig. 2.3. Log the physical trajectory $\{Y_i\}$, user input trace $\{U_h\}$, and code trace θ.

Task2: Run the pre-trained transformer model in parallel with the control-CPS simulate. For each new physical trajectory sample Y_{i+1} from the simulation, calculate the transformer prediction error e_{i+1} via Exp. 2.4 and log it.

Step2

When the simulation ends, check if the transformer prediction error trace $\{e_i\}$ contains $outlier(s)$ [34] [7]. If so, label θ as "incorrect"; otherwise label θ as "correct".

Step3

If enough code traces are collected, terminate; otherwise go to Step1.

Figure 2.4: Transformer Oracle Solution

Step1

Task1: Same as Fig. 2.4-**Step1-Task1**

Task2: Run AR-SI in parallel with the control-CPS simulation. For each new physical trajectory sample Y_{i+1} from the simulation, calculate the AR-SI prediction error e_{i+1} via Exp. 2.4 and log it.

Step2

Same as Fig. 2.4-**Step2**

Step3

Same as Fig. 2.4-**Step3**

Figure 2.5: AR-SI Oracle Solution

SFL tools: Tarantula [44], Crosstab, and Ochiai [9]. All oracles are implemented in Python. Guided by Table 2.1, we designed 27 bugs to inject into the ArduCopter cyber subsystem. Totally, we generate 37 versions (i.e. 37 subjects) of buggy ArduCopter systems. Each of the 27 buggy subjects contains one of the candidate artificial bugs and each of the rest 10 subjects contains 3 candidate artificial bugs.

2.4.1 Evaluation Method

For each buggy subject, we generate 10000 code traces from the ArduCopter emulation platform. The traces will be labeled by oracle, and then sent to three SFL tools: Tarantula, Crosstab and Ochiai (TR, CR and OI). In later evaluations, for example, TO-TR stands for Transformer Oracle with Tarantula for SFL, while AO-CR stands for AR-SI Oracle with Crosstab for SFL.

We use box plots to show the experimental results clearly. A box plot, also known

Table 2.1: Common bugs-in-the-field [65]

Type	**Description**
WPFV	Wrong Variable used in Parameter of Function call
WVAV	Wrong Value Assigned to Variable
MVAE	Missing Variable Assignment using Expression
MFC	Missing Function Call
MIA	Missing IF construct Around statements
MVIV	Missing Variable Initialization using a Value
MVAV	Missing Variable Assignment using a Value
MIFS	Missing IF construct plus Statements
MIEB	Missing IF construct plus statements plus ELSE Before statements
MLC	Missing a Logic Clause in branch condition
MLPA	Missing small and Localized Part of the Algorithm
WAEP	Wrong Arithmetic Expression in Parameter of function call

as a box and whisker plot, is a standardized way of displaying the dataset based on the five-number summary: the minimum, the maximum, the sample median, and the first and third quartiles [3]. We compare TO and AO for the control-CPS testbed ArduCopter from the following aspects.

Following the design evaluation metrics of [35], we choose accuracy and latency as our evaluation metrics. SFL tools will give the users a suspect list $S = (s_1, s_2, \cdots, s_l)$, where $s_i (i = 1, \cdots, l)$ is the ith suspected (s_1 is the most suspected) source code block. Given that B is the set of truly buggy blocks. Then *accuracy* refers to $|\{s_i | s_i \in B\}|/l$; *latency* refers to $min\{i | s_i \in B\}$.

RQ1: SFL quality. How do TO and AO impact the quality of SFL from the perspective of accuracy and latency?

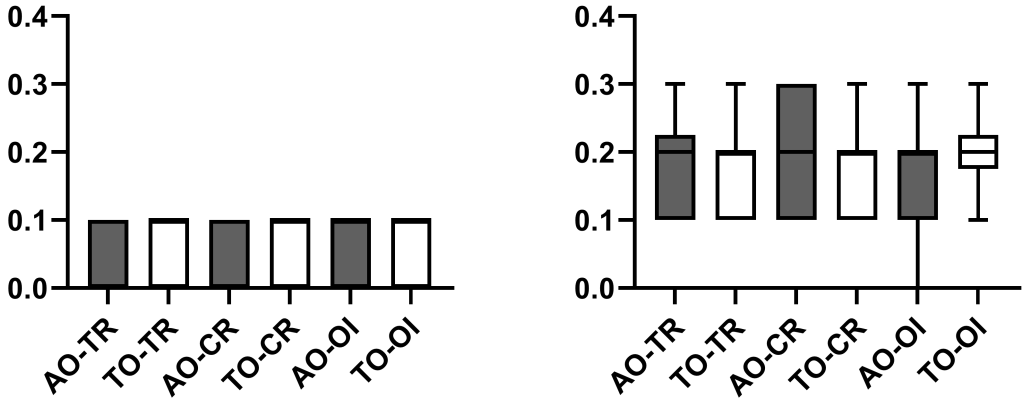
RQ2: Raw oracle quality. How do TO and AO impact the quality of raw oracle from the perspective of false positive rate and false negative rate?

RQ3: Significance of differences. What is the significance of the differences between TO and AO through T-test?

Study 1: SFL quality.

The results related to accuracy and latency are plotted in Fig.2.6 and Fig.2.7 respectively. From the perspective of accuracy, our results demonstrate that TO and AO perform similarly in 1-bug subjects' trials. In 3-bug subjects' trials, with TR and CR, AO seems to have a slightly better performance, while with OI, TO is slightly better than AO. From the perspective of latency in Fig.2.7, it is clear that TO has a slightly better performance than AO. For 1-bug subjects' trials, the range of TO and AO are all from 0 to 10. Besides, the median of TO is smaller than AO in each of three SFL tools, which indicates that TO can find bugs slightly faster in some cases than AO. For 3-bug subjects' trials, TO has a small advantage over AO with CR and

OI.



(a) 1-bug subjects' trials

(b) 3-bug subjects' trials

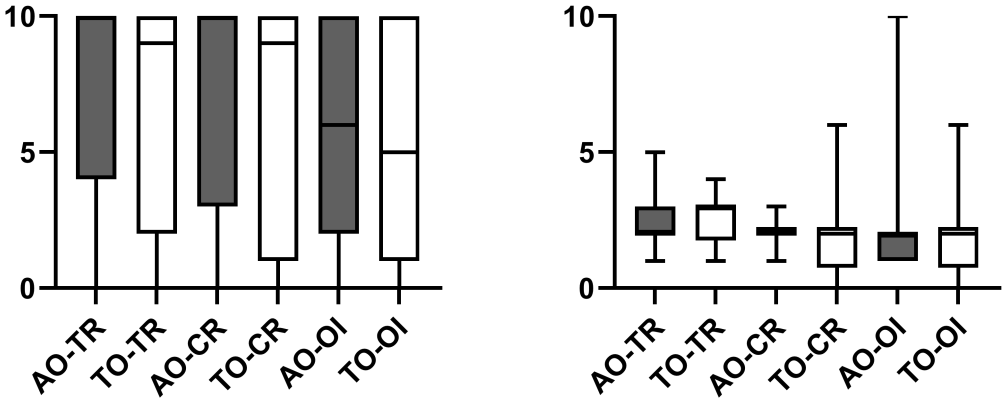
Figure 2.6: Accuracy (the higher the better)

Answer to RQ1: AO and TO perform similarly in SFL quality, but TO has a slightly better performance than AO in terms of latency (not significant).

Study 2: Raw oracle quality.

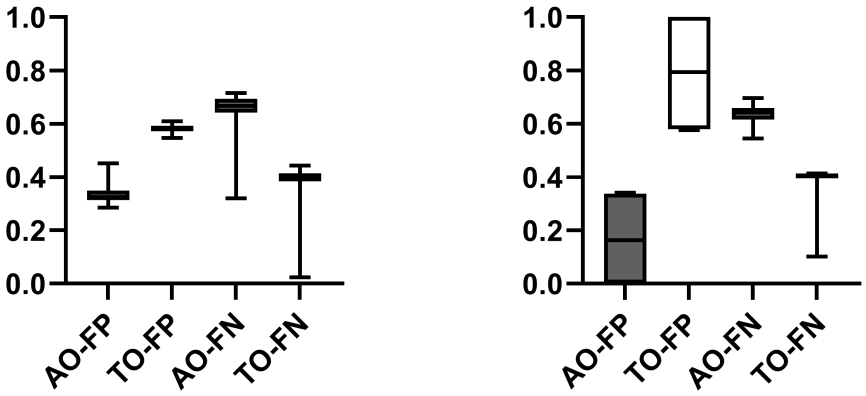
The result related to the false positive rate and the false negative rate is plotted in Fig.2.8. The labels on X-axis represent the oracle and relative quality metrics. FP: false positive rate. FN: false negative rate. For example, TO-FP stands for transformer oracle false positive rate. It is obvious that AO has a lower false positive rate and TO has a lower false negative rate, which implies that TO is more willing to label the traces as buggy.

Answer to RQ2: AO has a lower false positive rate and TO has a lower false negative rate.



(a) 1-bug subjects' trials (b) 3-bug subjects' trials

Figure 2.7: Latency (the lower the better)



(a) 1-bug subjects' trials (b) 3-bug subjects' trials

Figure 2.8: Oracle false positive rate and false negative rate

Study 3: Significance of differences.

The significance of TO and AO differences are quantified by the p-values [5] [25] and effect size (ES) values [4] in Table 2.2. P-values can measure the probability that an observed difference could have occurred just by random chance. The lower the p-value, the greater the statistical significance of the observed difference. Effect size can compare the magnitude of differences. We calculate by Cohen's d [98], a measure relating the mean difference to variability. We regard the absolute value of effect size over 0.4 as at least medium difference magnitude. As shown in Table 2.2, except for three cells in gray, the p-values are mostly above 5%, and except four cells in gray, the absolute value of effect size are below 0.4, hence there is no significant difference between TO and AO.

Table 2.2: Quality of AO vs TO

Metric		**1-bug subjects**		**3-bug subjects**	
		p-value	**ES**	**p-value**	**ES**
Accuracy	**TR**	10.3%	-0.15	67.8%	0.08
	CR	18.5%	-0.11	8.1%	0.21
	OI	32.7%	-0.04	19.3%	-0.21
Latency	**TR**	13.0%	0.14	83.2%	-0.05
	CR	23.1%	0.12	71.6%	0.08
	OI	27.5%	0.06	57.6%	0.14
FPR		<0.1%	-0.98	6.4%	-0.86
FNR		<0.1%	0.91	<0.1%	0.84

Answer to RQ3: There is no significant difference between AO and TO in terms of accuracy and latency.

2.4.2 Transformer Training

The structure of transformer basically follows the design of [93]. We generated 3000 trajectories using ArduCopter3.6.10 (a release version). Each trajectory consists of 240 Y_i', we can organize these data into 230 (T_q, T_{q+1}) groups. In total, we got 690,000 pairs of training data. We trained our transformer model with these data for 30 epoches. We chose MSEloss [6] as our loss function. After 30 epoches of training, the loss dropped from 58.96 at the beginning and stabilized at around 0.2.

2.5 Related Work

In this paper we focus on automatic oracle tools for control-CPS. The key is to identify whether the output trajectory of control-CPS is inline with user's expect (user's input). Using AR-SI as the automatic oracle for control-CPS [35] is a successful attempt, it uses AR-SI and sliding window to predict trajectory and then check outlier. Another successful trial is Mithra [11], using mature 3-step anomaly detection as control-CPS oracle. The result shows it has better performance than AR-SI approach. However, implementing Mithra requires more complicated steps than AR-SI, AR-SI is more convenient and thus more universal.

Metamorphic testing [23] is also very efficient as an oracle. The main idea is using the metamorphic relationships (the relationship between the software input change and output change during multiple program executions) as the oracle. Metamorphic testing was used in [51] to detect faults in embedded software. It's also an effective solution for oracle problems.

2.6 Conclusion

In this short paper, we proposed an automatic oracle method based on transformer. We compared the transformer oracle (TO) and AR-SI oracle (AO) on classic control-CPS test-beds with SFL of injected bugs. The results show, TO and AO perform similarly in SFL quality, but TO has a slightly better performance than AO in terms of latency; AO has a lower false positive rate and TO has a lower false negative rate. As for overall performance, there is no significant differences between TO and AO. AO can achieve good performance without pre-training, and it is very convenient to use. In order for TO to defeat AO, it may need better design or more training data. New research opportunities may include combining the physical laws with user input to make the transformer oracle's judgments more accurate.

Chapter 3

Fuzzy Logic Supported Intended Policy Variation: An Oracle for Testing Reinforcement Learning Software

3.1 Introduction

Nowadays, *data-driven AI machine learning* (referred to as "*AI machine learning*" in the following for simplicity) is being rapidly adopted across various domains [45] [104]. However, as the complexities of AI machine learning software soars, it encounters growing testing challenges [78] [19]. The high software complexities necessitate test automation; and test automation needs a critical tool called *oracle*. Given a set of test cases and software execution outputs, an oracle is the tool that automatically labels the correctness of the execution outputs, hence labels the correctness of the execution, which further implies if the software is "*buggy*" (i.e. contains bugs) or "*bugless*." It is well-known that oracles can be extremely difficult to design, depending

on the application domain. In fact, for many application domains, we do not yet have good enough oracles, hence "*human oracles*" (i.e. ask human experts to manually label the outputs' correctness) are still the mainstream practice [15], blocking the full automation of software testing. This is the so-called the *oracle problem* [15].

For the application domain of AI machine learning, the oracle problem is even bigger. It faces a unique challenge: how to define whether the output of a machine learning software is correct or wrong. For example, an AI machine learning software for facial recognition outputs a *Neural Network* (NN). How to judge whether this NN is correct or wrong? All that we can do is to throw a test set (a collection of pictures involving human faces and other contents) to the NN, and the NN outputs a set of recognition decisions. Even if there are wrong decisions (recognizing a human as car, or the reverse), it does not mean the AI machine learning software is buggy; and vice versa.

To address this problem for generic AI machine learning is too big a challenge. In this chapter, we are going to focus on *Reinforcement Learning* (RL) [87] [97] [85]. RL is a promising sub-category of AI machine learning [57] [62]. As illustrated by Fig. 3.1, an RL software trains an *agent* (e.g. an NN to autopilot a car) to interact with an *environment* (e.g. the actual physical environment of the car auto-piloting, or more commonly, the simulator of the physical environment). During the training stage, the agent continuously monitors the *state* of the environment (e.g. the position of the car). Given a state of the environment, the agent decides an *action* to be applied to the environment (the decision making *policy* is part of the agent). Each *action* applied by the agent to the environment will change the state of the environment, and trigger a *reward* to be fed back to the RL software. The RL software will modify the agent (particularly, the agent's policy), to pursue maximum cumulative rewards [47]. The final output of the training stage (by the RL software) is the trained agent.

RL naturally matches the underlying demands of a wide range of applications that pursue rewards, such as video-game [54], recommendation systems [105], and robotics (optimal control) [50]. In turn, the increasing demands for RL also foster more

complex RL software. The growing complexities of the RL software demands test automation, hence testing oracles. As a mainstream sub-category of AI machine learning, the oracle problem on testing AI machine learning software also stands out in RL.

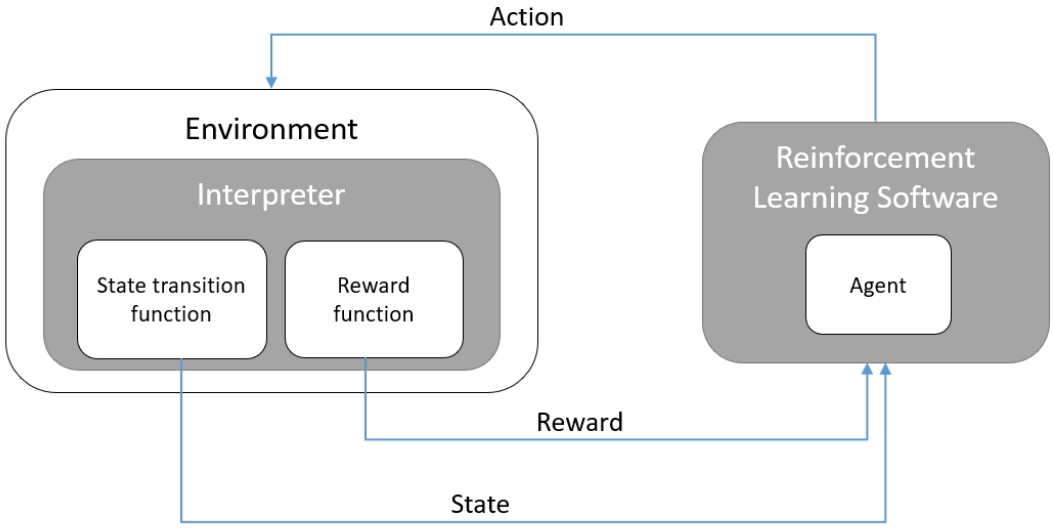


Figure 3.1: RL Framework

Unfortunately, the RL oracle problem is understudied. To our best knowledge, human oracles are still the mainstream for testing RL software. To better automate the testing, in this chapter, we are going to propose a non-human oracle, which exploits the features of RL. Specifically, we design our oracle upon the following two heuristics.

Heuristic 1: Behavioral Compliance to the Intended Policy

One feature of RL is that the environment always contains a *reward function* (see Fig. 3.1), which rewards "good" actions and punishes "bad" actions depending on the current state of the environment. The reward function must reflect certain underlying "*intended policy*," which typically only defines the *ideal actions* (aka "*reference*

actions") for a few *typical states* (aka "*reference states*"), but fuzzy on other cases. We propose to use fuzzy logic [48] to cover the other cases, so that a reward function can be defined to reflect the intended policy.

If an RL software implementation is bugless, under the stimulation of the aforementioned reward function, it should figure out and try to converge to the ground truth intended policy [87] [97] [85]: throughout E *epochs* of training (in this chapter, an *epoch* of RL consists of a series of consecutive *state-action interaction steps*, which generates a *learnt policy* to the user by its end), the E learnt policies shall show a trend of becoming compliant to the intended policy.

Based on this feature, before the training starts, we propose to randomly generate a set of I intended policies (i.e. the so-called "*variated* intended policies," or "*intended policy variation*") and the corresponding reward functions with the help of fuzzy logic. For each intended policy, we carry out E epochs of RL training of the agent. By the end of each training epoch, we use fuzzy logic [16] [42] again to quantify the learnt policy's compliance to the given intended policy. The resulted *epoch-compliance-value* time-series should show an increasing compliance trend throughout the E epochs (note the epoch-compliance-value is different from RL training loss: 1) only one epoch-compliance-value is generated per epoch; but K loss values are generated in an epoch involving K back-propagation steps; 2) epoch-compliance-value is based on the *ground truth* intended policy, while loss is based the RL software's understanding/estimation of the reward function). If not, the RL software is suspicious, and our oracle shall label it as "buggy."

Heuristic 2: Behavioral Abnormality after Convergence

The epoch-compliance-value time-series mentioned in **Heuristic 1** may or may not converge. However, if it shows a tendency of convergence, an RL software normally should not drastically damage this tendency (specifically, drastically reduce the epoch-compliance-value for many consecutive epochs) [17] [32]. We also exploit this feature

as part of our oracle.

Guided by the above heuristics, this chapter makes the following contributions:

1) We propose an oracle that exploits the features of RL software for their testing.

2) We compare our proposed oracle with the conventional human oracle, the result shows that our proposed oracle can outperform the human oracle in some test beds, and achieve comparable performance in other test beds. This means, when the human oracle is unavailable, people can use our proposed oracle to test RL software.

The rest of this chapter is structured as follows: Section 3.2 provides background on RL and fuzzy logic. Section 3.3 proposes our oracle. Section 3.4 evaluates our proposed oracle. Section 3.5 elaborates related work. Finally, Section 3.6 discusses future work and concludes the chapter.

The source code is accessible at our GitHub repository [73] RL-testing-new.

3.2 Background

3.2.1 Fuzzy Logic

Fuzzy logic is a method designed for dealing with imprecision. It allows computers to understand and manipulate vague concepts. It was introduced by Lotfi Zadeh in 1965 to overcome the limitations of binary (true or false) logic when dealing with incomplete or imprecise information [103] [49].

Fuzzy logic is built upon the concept of *fuzzy membership*, which quantifies the degree or likelihood of a particular element being part of a set. Unlike classical set theory, where an element either belongs or does not belong to a set, fuzzy membership allows more levels of belongings. This is quantified by a *fuzzy membership function*, typically

denoted as μ , which assigns each element a membership value between 0 and 1, where 0 represents complete non-membership and 1 represents full membership. The values in between reflect partial membership, providing a mathematical way to capture fuzzy concepts in natural language, such as "warm," "slow," or "high." Fig 3.2 exemplifies a fuzzy membership function for "warm."

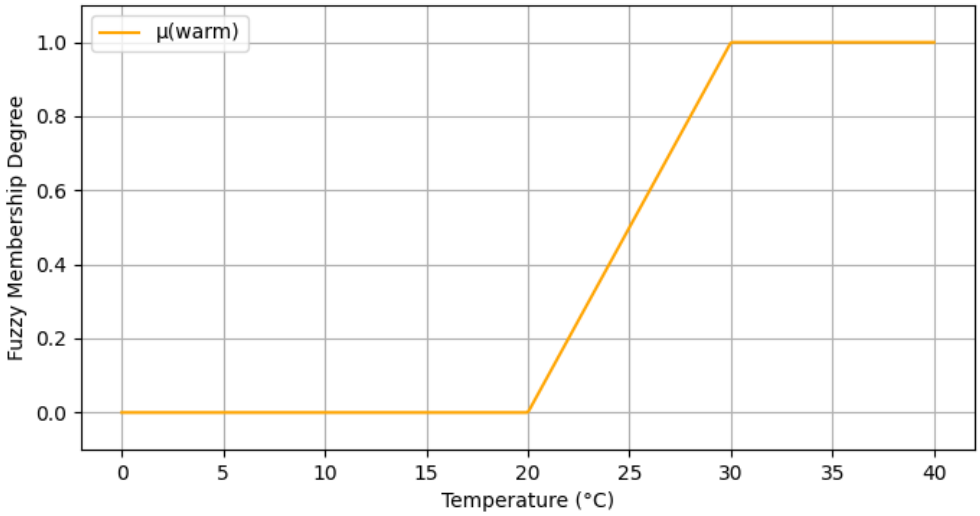


Figure 3.2: Fuzzy Membership Function of "Warm"

3.2.2 Reinforcement Learning (RL)

An RL framework, as shown by Fig. 3.1, has the following components [87]:

- **Environment:** This is the target system that we want to maneuver, typically denoted as $\mathcal{E}$. At any time instance, the environment has a *state* (explained later). The state changes when the environment takes as input an *action* (explained later). The environment also has a *reward function* (explained later) to evaluate the state change and the action. Then the environment outputs the new state and the new reward.

- **State:** This is a value, typically an n-dimensional vector and denoted as x (e.g. $x \in \mathbb{R}^n$), that defines the current situation of the environment. The set of all possible values for x, aka the *state set*, is denoted as X.

- **Action:** This is an input to the environment, typically an m-dimensional vector and denoted as a (e.g. $a \in \mathbb{R}^m$). The set of all possible values for a, aka the *action set*, is denoted as A.

- **State Transition Function:** A function within the environment that maps the current state x and the current inputted action a to a new state x'.

- **Reward Function:** A function within the environment that evaluates the impact of inputted action a upon the current state x. Typically the reward function is denoted as $R : X \times A \mapsto \mathbb{R}$. The output of the reward function, is also called the *reward value*, or simply the *reward*.

- **Interpreter:** The state transition function and the reward function are collectively called the *interpreter*.

- **Agent (and State-Action Interaction Step):** An entity that interacts with the environment. It consists of an internal function, aka *policy*, denoted as $\pi : X \mapsto A$. The policy takes as input the state $x \in X$ outputted from the environment, and outputs an action $a \in A$ to be sent to the environment. We call x and the corresponding a a "*state-action interaction step*," or simply "*state-action step*," denoted as (x, a).

- **RL Software:** The machine learning software to run during training stage to construct the agent. During the training stage, the RL software forwards the state (action) outputted from the environment (agent) to the agent (environment). Meanwhile, it modifies the agent (more specifically, the policy in the agent) based on the rewards outputted from the environment, and the objective is to maximize cumulative rewards.

3.3 Solution

Given a piece of RL software implementation $\mathcal{S}$ (simplified as "*RL implementation*" in the following) to test, Algorithm 1 formally specifies our proposed oracle, which realizes Heuristic 1 and 2 proposed in Section 3.1.

This algorithm takes as input the following parameters:

- $\mathcal{S}$: the RL implementation to be tested.
- $\mathcal{E}$: the environment to be connected to $\mathcal{S}$ to conduct testing. We can choose classic environments for benchmarking RL algorithms. For example, environments from the well-known Gymnasium [28] benchmark.
- I : the number of intended policies to generate. The exact configuration is discussed in Section 3.4.3.
- E : the number of training epochs for each intended policy. The exact configuration is discussed in Section 3.4.3.
- J : the number of state-action interaction steps for each epoch. The exact configuration is discussed in Section 3.4.3.
- θ_{orcl} : the *oracle threshold* to judge whether $\mathcal{S}$ is buggy; $\theta_{\mathsf{orcl}} \in [0, 1]$. The exact configuration is discussed in Section 3.4.5.

The algorithm consists of four sequential phases: initialization phase, training phase, log analysis phase, and oracle phase. The following subsections shall describe them one by one.

Algorithm 1. Our Proposed Oracle for Testing RL Implementation

1. **function** OracleMain (
   $\mathcal{S}$: the to-be-tested RL implementation,
   $\mathcal{E}$: the environment,
   I: the number of intended policies to generate,
   E: the number of training epochs for each intended policy,
   J: the number of state-action interaction steps for each epoch,
   $\theta_{\mathsf{orcl}} \in [0, 1]$: the oracle threshold to judge whether $\mathcal{S}$
   is buggy):

2. // Initialization Phase:
3. Generate a set of intended policies $\Pi_{\mathsf{ref}} = \{\pi_{\mathsf{ref},i}\}_{i=1,\ldots,I}$ and
   the corresponding set of reward functions $\mathcal{R} = \{R_i\}_{i=1,\ldots,I}$;
4. // An intended policy $\pi_{\mathsf{ref},i}$ is a function $X_{\mathsf{ref},i} \mapsto A_{\mathsf{ref},i}$, where
5. // $X_{\mathsf{ref},i} \subseteq X_{\mathcal{E}}$ is the reference state set, $A_{\mathsf{ref},i} \subseteq A_{\mathcal{E}}$ is the
6. // reference action set, $X_{\mathcal{E}}$ is the environment
7. // dependent state set, and $A_{\mathcal{E}}$ is the environment dependent
8. // action set. The creation of R_i also needs the help of
9. // other environment dependent functions and constants,
10. // see Eq. (3.1), (3.3), and (3.4).

11. // Training Phase:
12. **for** $(i : 1 \leqslant i \leqslant I)$ {
13. **for** $(e : 1 \leqslant e \leqslant E)$ {
14. // The eth training epoch:
15. $L_{i,e}$:= execute the RL implementation $\mathcal{S}$
   with the environment $\mathcal{E}$ and the reward function R_i
   (which reflects the intended policy of $\pi_{\mathsf{ref},i}$)
   as per the framework of Fig. 3.1;

16. // $L_{i,e}$ is the log of J state-action interaction steps
17. // defined as per Eq. (3.5).
18. }
19. }

20. // Log Analysis Phase:
21. Initialize the trends value set $\mathcal{T} := \varnothing$;
22. **for** $(i : 1 \leqslant i \leqslant I)$ {
23. Initialize the epoch-compliance-value time-series $\mathcal{M}_i := [\,]$;
24. **for** $(e : 1 \leqslant e \leqslant E)$ {
25. Append the epoch-compliance-value $\mu_{\mathsf{epoch}}(L_{i,e}, \pi_{\mathsf{ref},i})$ to $\mathcal{M}_i$;
26. // μ_{epoch} is defined as per Eq. (3.6) and (3.7)
27. }
28. Conduct linear regression on $\mathcal{M}_i$, and find the resulted slope α;
29. **if** $(\alpha > 0)$ **then** $\tau_i :=$ "true";
30. **else** $\tau_i :=$ "false";
31. Add τ_i as an element to the set $\mathcal{T}$;
32. }

33. // Oracle Phase:
34. $c :=$ number of "true" values in $\mathcal{T}$;
35. **if** $c/|\mathcal{T}| \geqslant \theta_{\mathsf{orcl}}$ { **return** "bugless"; } **else** { **return** "buggy"; }

3.3.1 Initialization Phase

In the initialization phase, the algorithm generates a set of I intended policies, denoted as $\Pi_{\mathsf{ref}} = \{\pi_{\mathsf{ref},i}\}_{i=1,\cdots,I}$, and the corresponding set of reward functions $\mathcal{R} = \{R_i\}_{i=1,\cdots,I}$ (see line 3 of Algorithm 1).

The notion of intended policy depends on the notion of *policy*. Given an environment $\mathcal{E}$ with state set $X_{\mathcal{E}}$ and action set $A_{\mathcal{E}}$, a *policy* is a function $\pi : X_{\mathcal{E}} \mapsto A_{\mathcal{E}}$. An *intended policy*, denoted as $\pi_{\mathsf{ref}} : X_{\mathsf{ref}} \mapsto A_{\mathsf{ref}}$, is a subset of a policy. It focuses on a set of *typical states* (aka *reference states*), denoted as X_{ref} ($X_{\mathsf{ref}} \subseteq X_{\mathcal{E}}$), and a set of *ideal actions* (aka *reference actions*), denoted as A_{ref} ($A_{\mathsf{ref}} \subseteq A_{\mathcal{E}}$), and maps X_{ref} to A_{ref}.

Given an intended policy $\pi_{\mathsf{ref}} : X_{\mathsf{ref}} \mapsto A_{\mathsf{ref}}$, the corresponding reward function $R : X_{\mathcal{E}} \times A_{\mathcal{E}} \mapsto \mathbb{R}$ shall define a reward value for any *state-action interaction step* $(x, a) \in X_{\mathcal{E}} \times A_{\mathcal{E}}$. We can manually do this when $|X_{\mathcal{E}}|$ and $|A_{\mathcal{E}}|$ are finite and small. Otherwise, how to define R becomes a challenge.

We propose to exploit fuzzy logic to define R.

Specifically, given the intended policy $\pi_{\mathsf{ref}} : X_{\mathsf{ref}} \mapsto A_{\mathsf{ref}}$ and arbitrary state-action interaction step $(x, a) \in X_{\mathcal{E}} \times A_{\mathcal{E}}$, we first find the closest (ties, if any, are broken randomly) $x^*_{\mathsf{ref}} \in X_{\mathsf{ref}}$ to x:

$$x^*_{\mathsf{ref}} \stackrel{\mathsf{def}}{=} \mathsf{argmin}_{x_{\mathsf{ref}} \in X_{\mathsf{ref}}} \{||x - x_{\mathsf{ref}}||_{\mathcal{E}}\}, \tag{3.1}$$

where $||\cdot||_{\mathcal{E}}$ is the environment dependent distance metric (e.g. Euclidean distance).

We quantify the similarity between x and x^*_{ref} with a *state-similarity-value* $\mu^{\mathcal{E}}_{\mathsf{state}}(x, x^*_{\mathsf{ref}})$, where $\mu^{\mathcal{E}}_{\mathsf{state}}$ is an environment dependent function of $X_{\mathcal{E}} \times X_{\mathsf{ref}} \mapsto [0, 1]$.

Denote

$$a^*_{\mathsf{ref}} \stackrel{\mathsf{def}}{=} \pi_{\mathsf{ref}}(x^*_{\mathsf{ref}}). \tag{3.2}$$

We also quantify the similarity between a and a^*_{ref} with an *action-similarity-value* $\mu^{\mathcal{E}}_{\mathsf{action}}(a, a^*_{\mathsf{ref}})$, where $\mu^{\mathcal{E}}_{\mathsf{action}}$ is an environment dependent function of $A_{\mathcal{E}} \times A_{\mathsf{ref}} \mapsto [0, 1]$.

Intuitively, $\mu^{\mathcal{E}}_{\mathsf{state}}(x, x^*_{\mathsf{ref}})$ and $\mu^{\mathcal{E}}_{\mathsf{action}}(a, a^*_{\mathsf{ref}})$ are respectively the fuzzy logic partial membership of x and a to x^*_{ref} and a^*_{ref}.

The *step-compliance-value* $\mu_{\mathsf{step}}(x, a)$ is defined as

$$\mu_{\mathsf{step}}(x, a) \stackrel{\mathsf{def}}{=} \mu^{\mathcal{E}}_{\mathsf{state}}(x, x^*_{\mathsf{ref}})\mu^{\mathcal{E}}_{\mathsf{action}}(a, a^*_{\mathsf{ref}}). \tag{3.3}$$

Intuitively, $\mu_{\mathsf{step}}(x, a)$ reflects the fuzzy logic partial membership of (x, a) to the state-action interaction demanded by the intended policy $(x^*_{\mathsf{ref}}, a^*_{\mathsf{ref}} = \pi_{\mathsf{ref}}(x^*_{\mathsf{ref}}))$. Hence $\mu_{\mathsf{step}}(x, a)$ can be directly used as the reward function. However, the environment $\mathcal{E}$ may come with a default reward function $R^{\mathcal{E}}_{\mathsf{default}}(x, a) : X_{\mathcal{E}} \times A_{\mathcal{E}} \mapsto \mathbb{R}$, e.g. $R^{\mathcal{E}}_{\mathsf{default}}(x, a) \equiv -1$. When our intended policy π_{ref} is too fuzzy on the given state x, using $R^{\mathcal{E}}_{\mathsf{default}}$ may be a better choice. Considering this, we define the reward function as follows:

$$R(x, a) \stackrel{\mathsf{def}}{=} \begin{cases} R^{\mathcal{E}}_{\mathsf{default}}(x, a) & (\text{if } ||x - x^*_{\mathsf{ref}}||_{\mathcal{E}} \geqslant D^{\mathcal{E}}_{\mathsf{ref}}), \\ f^{\mathcal{E}}_{\mathsf{step}}(\mu_{\mathsf{step}}(x, a)) & (\text{otherwise}); \end{cases} \tag{3.4}$$

where $D^{\mathcal{E}}_{\mathsf{ref}} \in \mathbb{R}^{\geqslant 0}$ is an environment dependent distance threshold, aka the *reference state vicinity distance threshold* for ease of narration; $f^{\mathcal{E}}_{\mathsf{step}} : [0, 1] \mapsto \mathbb{R}$ is an environment dependent non-decreasing function, to provide us further flexibility in defining the reward function (e.g. we can scale/skew μ_{step} with it).

3.3.2 Training Phase

After the initialization phase, the training phase starts. For each of the I intended policy $\pi_{\mathsf{ref},i}$ $(i = 1, \cdots, I)$ and its reward function R_i generated by the initialization phase, we carry out E epochs of training. The eth $(e = 1, \cdots, E)$ epoch consists of J state-action interaction steps, which are logged as

$$L_{i,e} = \{(x_{i,e,j}, a_{i,e,j})\}_{j=1}^{J}. \tag{3.5}$$

3.3.3 Log Analysis Phase

After the training phase, for each intended policy $\pi_{\mathsf{ref},i} : X_{\mathsf{ref},i} \mapsto A_{\mathsf{ref},i}$ (where $i \in \{1, \cdots, I\}$, $X_{\mathsf{ref},i}$ is the *reference state set*, and $A_{\mathsf{ref},i}$ is the *reference action set*), we have E training logs $L_{i,e} = \{(x_{i,e,j}, a_{i,e,j})\}_{j=1}^{J}$ ($e = 1, \cdots, E$), each corresponds to one training epoch. For the eth ($e = 1, \cdots, E$) training epoch, we define the *epoch-compliance-value* $\mu_{\mathsf{epoch}}(L_{i,e}, \pi_{\mathsf{ref},i})$ as follows.

$$\mu_{\mathsf{epoch}}(L_{i,e}, \pi_{\mathsf{ref},i}) \stackrel{\mathsf{def}}{=} \frac{\sum_{(x_{i,e,j}, a_{i,e,j}) \in L'_{i,e}} \mu_{\mathsf{step},i}(x_{i,e,j}, a_{i,e,j})}{|L'_{i,e}|}, \tag{3.6}$$

where $\mu_{\mathsf{step},i}$ is the step-compliance-value function defined for the intended policy $\pi_{\mathsf{ref},i}$ (see Eq. (3.3)); while

$$L'_{i,e} \stackrel{\mathsf{def}}{=} \left\{(x_{i,e,j}, a_{i,e,j}) \middle| (x_{i,e,j}, a_{i,e,j}) \in L_{i,e} \text{ and } ||x_{i,e,j} - x^*_{\mathsf{ref},i,e,j}||_{\mathcal{E}} < D^{\mathcal{E}}_{\mathsf{ref}}\right\}; \tag{3.7}$$

where $x^*_{\mathsf{ref},i,e,j}$ is the closest reference state from $x_{i,e,j}$ defined by Eq. (3.1), $||\cdot||_{\mathcal{E}}$ is the environment dependent distance metric (see Eq. (3.1)), and $D^{\mathcal{E}}_{\mathsf{ref}}$ is the environment dependent reference state vicinity distance threshold (see Eq. (3.4)).

The time series of the E epoch-compliance-values $\{\mu_{\mathsf{epoch}}(L_{i,e}, \pi_{\mathsf{ref},i})\}_{e=1}^{E}$ are stored as $\mathcal{M}_i$. We then check $\mathcal{M}_i$, to see whether the E epochs of training show a trend to converge to the intended policy $\pi_{\mathsf{ref},i}$.

Our method is straightforward. We conduct linear regression on the time series data points of $\mathcal{M}_i$. We say there is a "*trend to converge*" if and only if the linear regression results in a positive slope (intuitively, the epoch-compliance-value is increasing through training).

3.3.4 Oracle Phase

We count the ratio of epoch-compliance-value time series $\{\mathcal{M}_i\}_{i=1}^{I}$ that have a trend to converge, and if this ratio is no less than a given threshold θ_{orcl}, we say the RL program is "bugless," and "buggy" otherwise.

Later, we will discuss how to configure θ_{orcl} in Section 3.4 through sensitivey study.

3.4 Evaluation

3.4.1 To-be-tested Bugless/Buggy RL Implementation

We use a well-known open-source RL algorithm library, the Stable Baselines3 (SB3) [74], to create our set of to-be-tested RL implementation. Built on top of PyTorch [72], SB3 provides well-trusted implementations of many state-of-the-art RL algorithms. Without loss of generality, our evaluations focus on three representative RL algorithms implemented by SB3, which are respectively *Deep Q-Networks* (DQN), *Advantage Actor-Critic* (A2C), and *Proximal Policy Optimization* (PPO). We regard the up-to-date SB3 implementations of DQN, A2C, and PPO in SB3 [2] as the "bugless" implementations of our to-be-tested RL implementation. Note this is a best effort practice, as we, as human beings, cannot know which implementation is absolutely bugless.

On top of the "bugless" implementations, we inject bugs to generate the "buggy" implementations. Specifically, we inject two kinds of bugs: artificial bugs and real-world bugs.

We designed the artificial bugs based on surveys of RL software bugs [67], Table 3.1 summarizes the buggy RL implementations injected with our designed artificial bugs[1].

Based on Table 3.1, we injected artificial bugs into the bug-less DQN, A2C, and PPO implementations of bugless RL software, to create various buggy implementations. Note the numbers of buggy RL implementations for different algorithms (DQN, A2C,

[1]This chapter focuses on hidden bugs, hence the non-hidden bugs (those show themselves via compilation warnings/failures and/or runtime expections/failures) listed in [67] are not listed in Table 3.1

Table 3.1: Buggy RL Implementations Injected with Artificial Bugs

Bug Type	Buggy RL Implementations Designed		
	DQN	A2C	PPO
Exploring the environment bugs	2	0	0
Model bugs	2	2	1
Tensor and inputs bugs	0	3	2
Training bugs	4	3	4
Updating network bugs	4	1	4

Note, each buggy RL implementation is injected with one artificial bug.

PPO) are not even. This is because whether we can inject a specific type of bug into a bugless implementation depends on the semantics of the bug and the algorithm. Some semantics are easy for bug injection, some are difficult, even impossible.

Meanwhile, we collect real-world bugs by manually examining the Github commit log of SB3 [75] [2], including SB3's ancestor repository: *Stable Baselines* (SB) [38]. Many real-world bugs recorded in the commit log might show themselves via compilation warnings/failures and/or runtime exceptions/failures. Hence, those bugs can be detected without oracle. Our manual examination finally collected 4 real-world hidden bugs listed in Table 3.2, 3 for DQN and 1 for PPO. Correspondingly, we create 3 buggy DQN and 1 buggy PPO implementations, each with one real-world bug.

Based on the above narrations, we create $(12 + 3) = 15$ buggy DQN implementations (12 with artificial bug, 3 with real-world bug), $(9 + 0) = 9$ buggy A2C implementations (9 with artificial bug, 0 with real-world bug), and $(11 + 1) = 12$ buggy PPO implementations (11 with artificial bug, 1 with real-world bug). Table 3.3 summarizes the total numbers of bugless/buggy RL implementations.

Table 3.2: Real-World Bugs

Bug ID	Description
RW1 (a DQN bug)	File: `stable_baselines3/common/on_policy_algorithm.py` Line 1: "`dones,`" should be "`self._last_episode_starts, # type: ignore[arg-type]`"; Line 2: "`#delete this line`" should be replaced with "`self._last_episode_starts = dones`".
RW2 (a PPO bug)	File: `stable_baselines3/ppo/ppo.py` Line: "`entropy_loss = -log_prob.mean()`" should be "`entropy_loss = -th.mean(-log_prob)`".

RW3 (a DQN bug)	File: `stable_baselines3/dqn/policies.py`
	Remove explicit feature extractor initialization and assignment:
	`self.features_extractor = ...,`
	`self.features_dim = ...,`
	`"features_extractor": ...,`
	`"features_dim": ....`
	Restore original:
	`net_args = self._update_features_extractor(...)`
	`return QNetwork(**net_args).to(self.device).`

RW4 (a DQN bug)	File: `stable_baselines3/common/on_policy_algorithm.py` Restore value computation block: "`with th.no_grad():`" "`# Compute value for the last timestep`" "`values = self.policy.predict_values(...)`".

Table 3.3: Number of Bugless/Buggy RL Implementations

# of Implementations	DQN	A2C	PPO
Bugless	1	1	1
Buggy	15	9	12

3.4.2 Testbed RL Framework

With the to-be-tested RL implementation (i.e. the bugless/buggy implementations prepared by Section 3.4.1) ready, we can now construct the RL framework of Fig. 3.1, to run the RL implementation to generate input/output logs for oracles to work on.

Particularly, we need to pin down the "environment" and its "reward function" for the RL framework.

Environment

We choose to reuse the environments from the well-known RL evaluation environment repository, Gymnasium [28]. Gymnasium includes tens of environments with customizable reward functions for us to choose. All of them are widely-used and well-maintained, so that we can empirically regard them as bugless (again as a best effort practice). Without loss of generality, we choose two representative environments: Frozen Lake [26] and MountainCar Continuous [27].

Frozen Lake demands a discrete decision-making agent (to be trained by the to-be-tested RL implementation). The agent shall navigate an avatar through a "frozen lake" chess-board-like environment to reach a target, and shall avoid the avatar from falling into "holes in the frozen lake." The Frozen Lake environment can be connected to all three RL algorithms: DQN, A2C, and PPO.

MountainCar Continuous plays with the digital twin of a continuous world mechanical

car. It demands the decision-making agent (to be trained by the to-be-tested RL implementation) to drive the car up a steep hill by building up enough momentum at the valley. As DQN cannot support continuous decision-making, the MountainCar Continuous environment can only be connected to two RL algorithms: A2C and PPO.

Therefore, the total numbers of feasible combinations of environment and RL implementation are summarized by Table 3.4.

Table 3.4: Number of Feasible Combinations of Environment and RL Implementation

# of Feasible Combinations	DQN	A2C	PPO
With Bugless RL Implementation	$1 \times 1 = 1$	$2 \times 1 = 2$	$2 \times 1 = 2$
With Buggy RL Implementation	$1 \times 15 = 15$	$2 \times 9 = 18$	$2 \times 12 = 24$

Note: $x \times y$ means there are x feasible environments for the corresponding RL algorithm, and y implementations for the corresponding bugless/buggy RL algorithm. See Table 3.3 for the values for y.

Reward Function

Both Frozen Lake and MountainCar Continuous allow us to customize the reward functions, so that we can realize line 3 of Algorithm 1. That is, we need to generate I intended policies $\pi_{\mathsf{ref},i}$ ($i = 1, \cdots, I$), and their corresponding reward functions R_i ($i = 1, \cdots, I$), and plug R_i into the environment $\mathcal{E}$.

To generate the I intended policies, for Frozen Lake, intended policy $\pi_{\mathsf{ref},i}$ ($i = 1, \cdots, I$) is generated as follows. We randomly sample $K = 8$ distinct states from the environment dependent state set $X_{\mathcal{E}}$ as the reference state set $X_{\mathsf{ref},i}$. For each sampled reference state $x_{\mathsf{ref},i,k}$ ($k = 1, \cdots, K$), we randomly sample a legitimate action

$a_{\mathsf{ref},i,k}$ from the environment dependent action set $A_{\mathcal{E}}$ as the intended action. That is $\pi_{\mathsf{ref},i}(x_{\mathsf{ref},i,k}) \stackrel{\mathsf{def}}{=} a_{\mathsf{ref},i,k}$. For MountainCar Continuous, intended policy $\pi_{\mathsf{ref},i}$ ($i = 1, \cdots, I$) is generated similarly. We randomly sample $K = 12$ sufficiently separated states from the environment dependent state set $X_{\mathcal{E}}$ as the reference state set $X_{\mathsf{ref},i}$. For each sampled reference state $x_{\mathsf{ref},i,k}$ ($k = 1, \cdots, K$), we randomly sample a legitimate action $a_{\mathsf{ref},i,k}$ from the environment dependent action set $A_{\mathcal{E}}$ as the intended action. That is $\pi_{\mathsf{ref},i}(x_{\mathsf{ref},i,k}) \stackrel{\mathsf{def}}{=} a_{\mathsf{ref},i,k}$.

With the intended policies $\pi_{\mathsf{ref},i}$ ($i = 1, \cdots, I$) ready, we follow the instructions of Section 3.3.1 to generate the corresponding reward function R_i.

Particularly, the environment dependent functions and constants mentioned in Section 3.3.1 are set as follows.

For measuring state distances $||x - x^*_{\mathsf{ref},i}||_{\mathcal{E}}$, we adopt the Euclidean distance.

For the environment of Frozen Lake, the state-similarity-value is defined as

$$\mu^{\mathcal{E}}_{\mathsf{state}}(x, x^*_{\mathsf{ref}}) \stackrel{\mathsf{def}}{=} \mathsf{exp}(-||\mathsf{loc}(x) - \mathsf{loc}(x^*_{\mathsf{ref}})||), \tag{3.8}$$

where $\mathsf{loc}(x)$ extracts state x's avatar location on the Frozen Lake chessboard; $||\cdot||$ measures the Euclidean distance; and exp is the exponential function.

For the environment of MountainCar Continuous, the state-similarity-value is defined as

$$\mu^{\mathcal{E}}_{\mathsf{state}}(x, x^*_{\mathsf{ref}}) \stackrel{\mathsf{def}}{=} \begin{cases} 1, & (\text{if } |\mathsf{dim0}(x) - \mathsf{dim0}(x^*_{\mathsf{ref}})| < 0.5); \\ 0, & (\text{otherwise}); \end{cases} \tag{3.9}$$

where $\mathsf{dim0}(x)$ extracts state x's 0th dimension value (intuitively, the car's location in the valley in the digital twin).

For the environment of Frozen Lake, the action-similarity-value is defined as

$$\mu^{\mathcal{E}}_{\mathsf{action}}(a, a^*_{\mathsf{ref}}) \stackrel{\mathsf{def}}{=} \begin{cases} 1, & (\text{if } a = a^*_{\mathsf{ref}}); \\ 0, & (\text{otherwise}). \end{cases} \tag{3.10}$$

For the environment of MountainCar Continuous, the action-similarity-value is defined as

$$\mu^{\mathcal{E}}_{\mathsf{action}}(a, a^*_{\mathsf{ref}}) \stackrel{\mathsf{def}}{=} \begin{cases} 1, & (\text{if } |a - a^*_{\mathsf{ref}}| < 0.3); \\ 0.7, & (\text{if } 0.3 \leqslant |a - a^*_{\mathsf{ref}}| < 0.5); \\ 0, & (\text{otherwise}); \end{cases} \tag{3.11}$$

where action a (as well as a^*_{ref}) is a signed scalar representing the force applied to the car.

With $\mu^{\mathcal{E}}_{\mathsf{state}}$ and $\mu^{\mathcal{E}}_{\mathsf{action}}$, we can calculate μ_{step} as per Eq. (3.3). With μ_{step}, we can now define the reward function as per Eq. (3.4), which, however, needs three more environment dependent functions/constants: $R^{\mathcal{E}}_{\mathsf{default}}$, $D^{\mathcal{E}}_{\mathsf{ref}}$, and $f^{\mathcal{E}}_{\mathsf{step}}$.

For Frozen Lake, we define the default reward $R^{\mathcal{E}}_{\mathsf{default}} \stackrel{\mathsf{def}}{=} -1$.

For MountainCar Continuous, we reuse its default reward function provided by the Gymnasium repository as the $R^{\mathcal{E}}_{\mathsf{default}}$.

For Frozen Lake, we define $D^{\mathcal{E}}_{\mathsf{ref}} \stackrel{\mathsf{def}}{=} 1$; for MountainCar Continuous, we define $D^{\mathcal{E}}_{\mathsf{ref}} \stackrel{\mathsf{def}}{=} 0.05$.

For Frozen Lake, we define $f^{\mathcal{E}}_{\mathsf{step}}$ as

$$f^{\mathcal{E}}_{\mathsf{step}}(\mu_{\mathsf{step}}) \stackrel{\mathsf{def}}{=} \begin{cases} 5\mu_{\mathsf{step}}, & (\text{if } \mu_{\mathsf{step}} > 0.7); \\ -0.5, & (\text{otherwise}). \end{cases} \tag{3.12}$$

For MountainCar Continuous, we define $f^{\mathcal{E}}_{\mathsf{step}}$ as

$$f^{\mathcal{E}}_{\mathsf{step}}(\mu_{\mathsf{step}}) \stackrel{\mathsf{def}}{=} 5\mu_{\mathsf{step}}. \tag{3.13}$$

3.4.3 Other Oracle Configurations

Besides the environment dependent functions/constants discussed in Section 3.4.2, there are four more configurable input parameters for our proposed oracle, i.e. Algo-

rithm 1: I, E, J, and θ_{orcl}.

We will discuss the configuration of the oracle threshold θ_{orcl} in Section 3.4.5. The configurations of I, E, and J are discussed as follows.

First, due to limited computational resources, we choose to generate only $I = 10$ intended policies for each environment and RL implementation combination.

Second, if $\mathcal{E}$ is the Frozen Lake environment, we set the number of training epochs E to 300, with each epoch consisting of $J = 200$ state-action interaction steps. For the more complex and challenging MountainCar Continuous environment, we set $E = 700$ and $J = 500$ to ensure sufficient training.

3.4.4 Comparison Baseline (Human Oracle)

To validate the effectiveness of our method, we set up a control group consisting of a human expert team to serve as the *human oracle*.

The human expert team is carefully selected to ensure a high level of expertise and diverse backgrounds. It includes five members: one member with a Ph.D. degree in control and computer science/engineering; one with a Bachelor's degree in computer science/engineering; one graduate student and two undergraduate students, all with sufficient research experience in the relevant fields.

We recreate conditions that closely resemble real-world RL software development for the human expert team. For each combination of the environment and RL implementation, we first record an animation of the untrained agent's behavior in the environment. After training, we record another animation of the trained agent's behavior in the environment. We also use Tensorboard [8], a widely used tool for NN evaluation, to log the rewards and losses throughout the RL training. Each human expert evaluates the pre- and post-training animations with the Tensorboard reward/loss logs holistically, and subjectively votes on whether the RL training is morbid, i.e.,

whether the RL algorithm implementation is buggy. The majority vote wins.

Note the total numbers of feasible combinations of environment and RL implementation are summarized by Table 3.4. However, due to logistic reasons (staff turnover), our human expert team is unable to finish evaluating the specific combination of MountainCar Continuous and PPO-with-real-world-bug implementation (there is only one PPO-with-real-world-bug implementation, see Table 3.2). To make fair comparisons between our proposed oracle and the human oracle, we have to remove the combination of MoutainCar Continuous and PPO-with-real-world-bug implementation from our evaluations. Therefore, the total numbers of actually evaluated combinations of environment and RL implementation are slightly adjusted from Table 3.4. The adjusted result is Table 3.5.

Table 3.5: Number of Actually Evaluated Combinations of Environment and RL Implementation

# of Actually Evaluated Combinations	DQN	A2C	PPO
Bugless	1	2	2
Buggy	15	18	$24 - 1 = 23$

3.4.5 Research Questions and Answers

We study the following research questions.

RQ1: How to set the oracle threshold θ_{orcl} (see line 35 of Algorithm 1) for our proposed oracle?

RQ2: How good is our proposed oracle compared to the baseline (i.e. the current mainstream practice of human oracle)?

The answers are elaborated as follows.

RQ1: How to set the oracle threshold θ_{orcl} for our proposed oracle?

In order to properly configure the oracle threshold θ_{orcl}, Fig. 3.3 plots the ROC curve of our proposed oracle with various θ_{orcl} configurations. The test set, as summarized by Table 3.5, consists of (1+2+2+15+18+23) = 61 combinations of environment and bugless/buggy RL implementation (i.e. 61 combinations of $(\mathcal{E}, \mathcal{S})$ for Algorithm 1).

Based on the ROC curves, we decide to set the oracle threshold θ_{orcl} to 0.6, which achieves the maximal *area under curve* (AUC). This choice allows us to maintain a relatively low *false positive rate* (FPR) while achieving a satisfactory *true positive rate* (TPR), hence striking a balance between the goal of minimizing false positives and the goal of identifying all the buggy implementations.

RQ2: How does the proposed oracle compare to the human oracle for different environments and algorithms?

To address this research question, again, we use the 61 combinations of environment and bugless/buggy RL implementation of Table 3.5 as the test set, and compare various performance metrics of our proposed oracle and the human oracle.

Table 3.6 and Fig. 3.4 compare the performance of our proposed oracle and the human oracle for different environments. We see that our proposed oracle apparently performs worse than the human oracle for the Frozen Lake environment, particularly in terms of *true positive* (TP) count, *false negative* (FN) count, accuracy, recall, and F1 score. However, for the MountainCar Continuous environment, the difference between our proposed oracle and the human oracle is not significant. This is probably because Frozen Lake is a discrete system, whose right or wrong are unambiguous; while MountainCar Continuous is a continuous system, whose right or wrong are

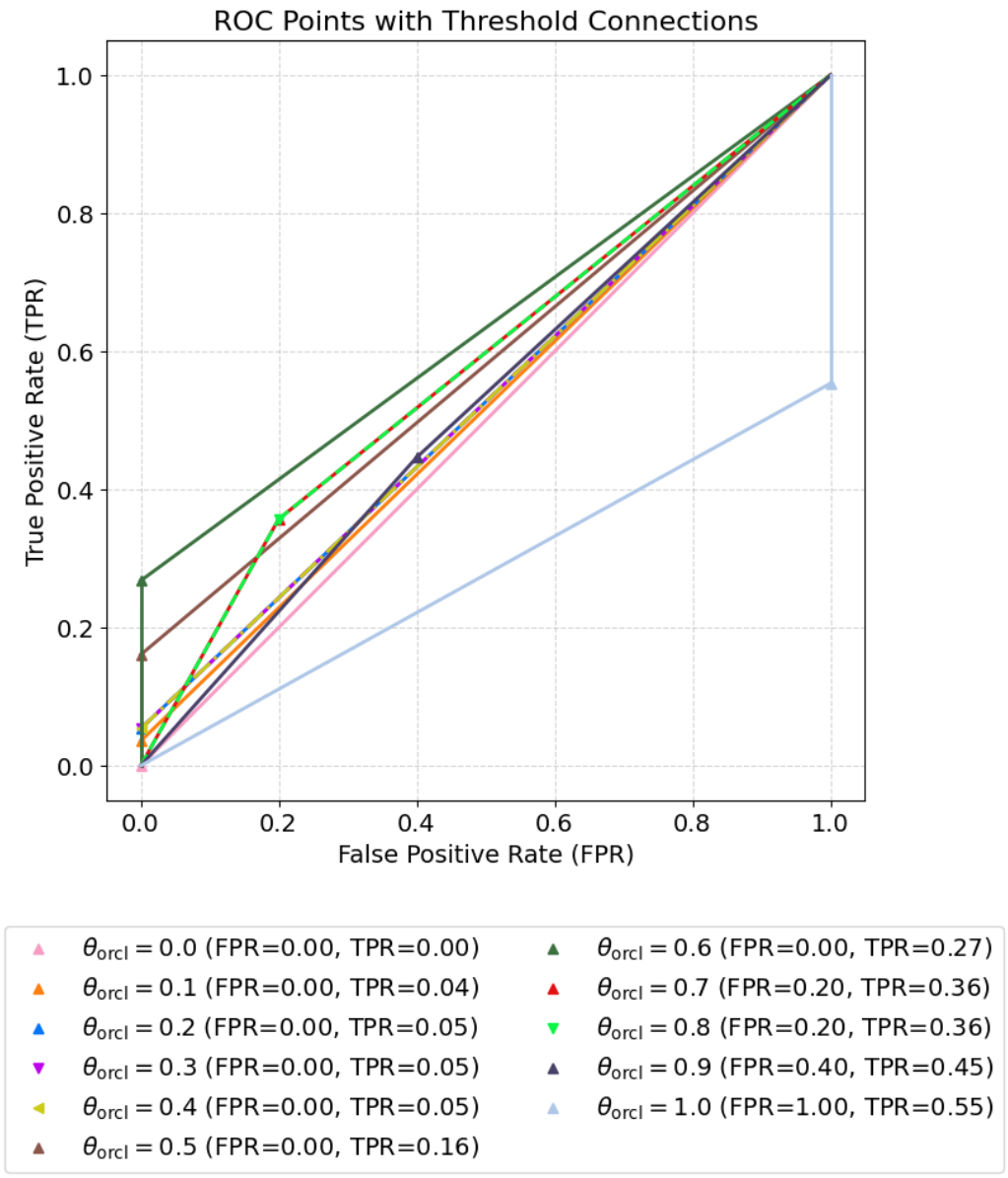


Figure 3.3: ROC Curve for Our Proposed Oracle with Different θ_{orcl} Values

more ambiguous (aka fuzzy). It is easier for human experts to judge systems with unambiguous definitions of right and wrong.

Table 3.6: Comparisons of Oracles for Different Environments

Oracle	**Environment**	**TP**	**FP**	**TN**	**FN**
proposed oracle	Frozen Lake	4	0	3	32
	Mountaincar Continuous	9	0	2	11
human oracle	Frozen Lake	23	1	2	13
	Mountaincar Continuous	8	0	2	12

TP: True Positive Count; FP: False Positive Count;

TN: True Negative Count; FN: False Negative Count.

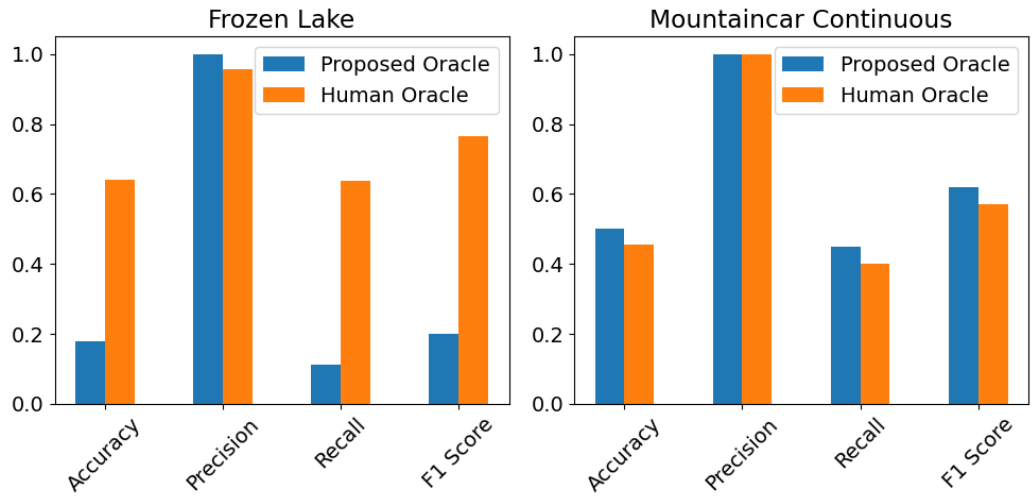


Figure 3.4: Comparisons of Oracles for Different Environments

Table 3.7 and Fig. 3.5 compare the performance of our proposed oracle and the human oracle for different RL algorithms. We see that our proposed oracle performs worse than the human oracle for the DQN and PPO algorithm for most metrics; while the performances on the A2C algorithm are the same. In addition, our proposed oracle performs equivalently well as or slightly better than the human oracle on the metric of precision.

Table 3.7: Comparisons of Oracles for Different RL Algorithms

Oracle	**Algorithm**	**TP**	**FP**	**TN**	**FN**
proposed oracle	A2C	8	0	2	10
	DQN	2	0	1	13
	PPO	3	0	2	20
human oracle	A2C	8	0	2	10
	DQN	13	1	0	2
	PPO	10	0	2	13

TP: True Positive Count; FP: False Positive Count;

TN: True Negative Count; FN: False Negative Count.

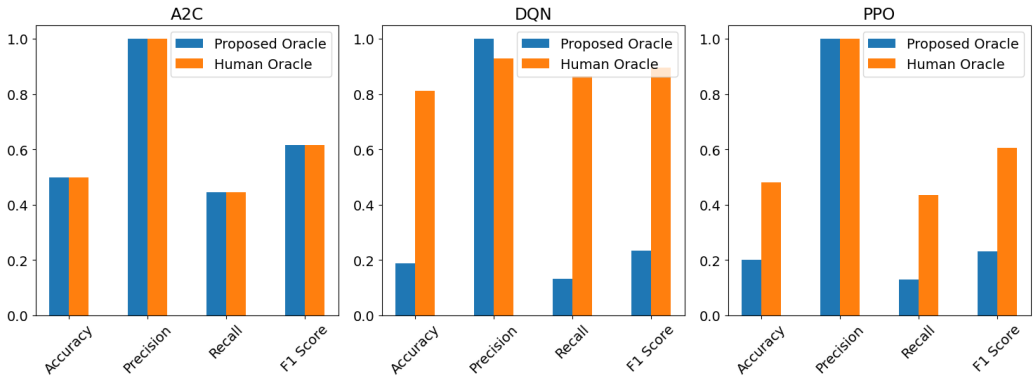


Figure 3.5: proposed oracle vs. human oracle in different Algorithms

From these comparisons, we see that our proposed oracle is not significantly better than the human oracle. This necessitates further study of the oracle problem for RL software.

We need to better exploit domain knowledge in the design of the oracle, to enhance the performances.

Also, from Section 3.3 and 3.4.2, we see that our proposed oracle involves many environment dependent functions and constants, which increases the realization difficulties. We want a better oracle with less environment dependency.

The above two demands inspired our work of Chapter 4.

3.5 Related Work

3.5.1 Fuzzy Logic Testing

Fuzzy logic has been applied in various aspects of software testing to handle uncertainty and provide more comprehensive evaluations. Researchers have explored the use of fuzzy logic in test case generation, test case prioritization, and test oracle construction.

In the area of test case generation, Bai et al. [14] proposed a fuzzy logic-based approach for generating test cases from software specifications. They used fuzzy set theory to model the input domain and employed fuzzy inference rules to determine the most effective test cases. This approach allowed for the generation of test cases that cover the input space more comprehensively and efficiently compared to traditional crisp methods. However, their work focuses on generating test cases for general software systems, while our approach specifically targets the testing of reinforcement learning software.

Fuzzy logic has also been used in test case prioritization to improve the effectiveness of testing efforts. Xu et al. [102] developed a fuzzy logic-based approach for prioritizing test cases in regression testing. They considered various factors, such as code coverage, fault detection history, and execution time, and used fuzzy inference rules to assign priority values to test cases. This approach helped in identifying the most critical test cases and optimizing the testing process. In contrast, our work focuses on evaluating the learning progress of reinforcement learning software rather than prioritizing test cases.

Wan et al. propose DRLFuzz [95], a coverage-guided fuzzing framework for testing

Deep Reinforcement Learning systems, focusing on generating diverse failure cases. While sharing the common goal of improving RL system reliability, our research differs by addressing challenges arising from the complexity and diversity of RL implementations. We introduce fuzzy logic for test oracle design and propose reward function variation to expand test coverage, while DRLFuzz employs coverage-guided fuzzing to automatically generate test cases. Despite the differences in approach, both studies tackle challenges like complexity and uncertainty in RL testing, and the ideas presented in DRLFuzz offer valuable insights for our research.

Test oracle construction is another area where fuzzy logic has been applied. Shahamiri et al. [82] proposed a fuzzy logic-based approach for constructing test oracles in the absence of explicit specifications. They used fuzzy inference rules to determine the expected behavior of the system based on its inputs and outputs. This approach provided a more flexible and robust way of constructing test oracles, especially in scenarios where precise specifications are not available. While their work deals with test oracle construction for general software systems, our approach specifically uses fuzzy logic to compare learned policies with intended policies in the context of reinforcement learning.

3.5.2 Reinforcement Learning Testing and Debugging

Reward-function variation has the potential to become a benchmark in the area of zero-shot Generalization of RL, yet it is very under-studied [48]. Similarly, we have not observed this promising technique being utilized within the test oracle designing of Reinforcement Learning. Nonetheless, we have identified some related work in the testing of RL:

DRLinter [67] proposed by Guo et al., is a rule-based static analysis tool for detecting potential faults in DRL software. DRLinter defines a set of rules based on expert knowledge and best practices to identify common antipatterns and bad practices,

such as suboptimal reward shaping, incorrect tensor operations, and inappropriate hyperparameter settings. In contrast, our work focuses on the dynamic testing of RL software by combining fuzzy logic and reward function variation to create a more robust test oracle. We aim to handle the ambiguity and uncertainty inherent in reward functions and explore a wide range of scenarios through reward function variations. While DRLinter and our research address different aspects of DRL system reliability, they complement each other in the pursuit of developing robust and dependable DRL applications, highlighting the importance of combining static analysis and dynamic testing techniques for comprehensive quality assurance.

STARLA [106] is a search-based testing approach for DRL agents that aims to find faulty execution epochs efficiently. It uses genetic algorithms to search the state-action space for sequences that are likely to trigger faults, guided by machine learning models trained on historical epoch data to predict the likelihood of faults. STARLA also incorporates state abstraction techniques to reduce the search space by clustering similar states based on the agent's perception, i.e., the optimal state-action value function. STARLA determines whether the model has triggered faults by identifying "faulty states". In STARLA, a faulty state is defined as a state that violates predefined requirements and is often an end state (termination state). For example, in an autonomous driving scenario, a state where the vehicle collides with an obstacle is considered a faulty state. However, STARLA differs from our proposed oracle method in its focus and granularity of analysis. While STARLA emphasizes detecting faults that occur in specific states within an epoch, our method focuses on analyzing the agent's learning process and progress over time by providing a continuous assessment of the agent's policy similarity trend during training. Combining STARLA's fine-grained, state-level fault detection with our continuous, learning-oriented analysis could potentially lead to a more comprehensive and explainable DRL testing approach, enabling us to identify faults in specific epochs and gain insights into the agent's evolving behavior throughout the training process.

MDP Playground [76] provides a way to analyze and debug RL agents by controlling various dimensions of hardness, allowing for experiments on toy environments and studying the effects on complex environments. It aims to improve agent performance and adaptability by examining factors such as non-Markov information states, irrelevant features, representations, and low-level dimensions of hardness. The difference between [76] and our approach is that MDP Playground provided a testbed, thus simplifies the creation of test environments for RL algorithms, but it still uses test set and evaluation metrics as RL test oracle. In contrast, Reward-function variation is a more universal and straightforward which does not rely on a test set or evaluation metrics.

Mohammad et al. [96] provides us a white-box testing approach for Deep Neural Networks. This paper introduces a method for detecting errors in deep neural networks by tracking and analyzing the flow of values during training, using imperative representations and probes for dynamic analysis to pinpoint faulty layers or hyperparameters, and an algorithm to identify root causes by monitoring numerical errors and the influence of each component on the model's performance. There are two main differences between this work and our reward-variation method: Firstly, RL algorithms are not always Deep Neural Networks, for example, Q-learning doesn't rely on Deep Neural Network. In that case, [96] may not work. Secondly, our Reward-function variation method treats the model as black box while [96] needs to dig into the model. Therefore, the two methods are suitable for different scenarios.

Maximilian et al. [43] introduces a novel regularization technique for RL algorithms—Selective Noise Injection (SNI) and Information Bottleneck (IB)—which can significantly enhance the generalization of agents in new environments. The SNI and IB methods are used during the training phase to enhance the generalization abilities of the model, while our method is used during the testing phase.

Gu et al. [33] proposed an approach for testing deep learning libraries by creating varied neural network structures to uncover bugs, focusing on ensuring the libraries'

reliability for tasks including training and inference phases. Muffin assesses foundational software integrity, our method works as test oracle for RL algorithms.

3.6 Conclusion

In this chapter, we investigated the oracle problem in testing RL software by proposing an oracle using fuzzy logic. Our proposed oracle exploits the inherent features of RL. First, if an RL software implementation is bugless, then under the stimulations of a proper reward function that reflects the intended policy (specifically, we exploited fuzzy logic to construct such reward function out of the intended policy), the RL software implementation shall figure out and converge to the intended policy. Second, if the RL software implementation is bugless, then during training, if the aforementioned convergence tendency appears, then RL software shall not drastically damage this tendency.

We evaluated our proposed oracle with human oracle against a test set of 61 combinations of environments and bugless/buggy RL software implementations. The results show that our proposed oracle is not significantly better than the human oracle, except for a few metrics.

This reveals the difficulty of the RL oracle problem, and piques us to exploit deeper domain knowledge in the design of the oracle, to enhance the performances, as well as simplifying the design. This directly fosters our work of Chapter 4.

Chapter 4

A Test Oracle for Reinforcement Learning Software based on Lyapunov Stability Control Theory

4.1 Introduction

Reinforcement Learning (RL) has gained significant attention in recent years due to its wide range of applications, such as control/robotics [50] and game playing [84]. RL software outputs an agent, which learns optimal decision-making policies through interacting with an inputted *training environment* (simplified as "*environment*" in the following); the agent's goal in the interaction is to maximize cumulative rewards [87]. As RL software grows more complex and infiltrates critical application domains, ensuring its quality and correctness becomes increasingly important.

One indispensable aspect of software quality/correctness insurance is testing. The

purpose of testing is to reveal the existence of bugs[1,2] in software. However, testing RL software faces unique challenges compared to testing traditional software.

To test traditional software, given the input, the correctness of an output is well defined: there is usually a unique and well-defined expected output, and the output is either equal to the unique expected output (i.e. correct), or not equal to it (i.e. incorrect). There is no gray area between the correct and the incorrect outputs. However, for RL software, given the inputted environment, often the expected output (i.e. the trained agent) is neither unique nor well-defined. For example, one major application domain of RL is to train agents (aka "*controllers*" in such contexts) for nonlinear control systems, where the nonlinearity is so complex, that the system is hard (if at all possible) to be mathematically analyzed. For such control systems, it is hard to define which controller is the unique correct controller (i.e. the unique well-defined expected output). All that we can measure is the controllers' performance for a test set; and the performance metrics are usually multi-dimensional, statistical, and dependent on the test set. It is difficult to define the optimum performance, nor to define what performance is correct/incorrect.

For example, in the control task of stabilizing an inverted pendulum [64], given a test set of initial states of the inverted pendulum, controller A on average stabilizes the inverted pendulum in 10 seconds, while controller B on average stabilizes the inverted pendulum in 15 seconds (usually a shorter stabilizing time cost is better). However, in terms of overshoot, controller A causes a worst case overshoot of 15 degrees, while controller B causes a worst case overshoot of 5 degrees (usually a smaller overshoot is

[1]A more formal term for "*bug*" is "software defect." But for succinctness, we will use the term "bug" in this chapter. Correspondingly, the formal terms "software contains defects" and "software contains no defect" will be referred to as "*buggy*" and "*bug-less*" respectively in the following.

[2]There are two types of bugs. Some bugs can cause compilation warnings/failures, or catchable runtime exceptions/failures (e.g. via the Java/C++ try-catch exception mechanisms, or the C error return values). Others are hidden. The hidden bugs are more threatening. Therefore, in the following, unless otherwise denoted, we shall focus on the hidden bugs.

better). Due to the multi-dimensionality of the performance metrics, it is difficult to define which controller is better, nor to define which controller is correct/incorrect: we cannot say controller A is incorrect because it performs worse than controller B on overshoot; nor controller B is incorrect because it performs worse than controller A on stabilization time cost.

The above hard-to-define-output-correctness challenge for RL software leads to a critical problem in testing: the *test oracle problem* [15].

A *test oracle* is an indispensable tool for software testing [15]. It is a mechanism to judge the correctness of software outputs (given the corresponding input) without analyzing the software source code (in the RL literature, the term "oracle" has other meanings; however, in this chapter, the term "oracle" never takes those meanings; therefore, in the following, we simplify the term "test oracle" as "oracle"). According to this definition, *the prerequisite to define an oracle is to define the correctness of software outputs*.

The RL software outputs are the agents (controllers). From the previous inverted pendulum example, we can see that to judge if an agent (controller) is correct, all that we can have is the agent's statistical performances (analytical proof of correctness is infeasible). However, as the inverted pendulum example shows, it is hard to use common statistical metrics to judge which agent is better, not to speak of which agent is correct/incorrect. That is, it is hard to define the correctness of RL software outputs. In other words, the prerequisite to define RL software oracle is yet to be fulfilled.

To deal with this problem, the traditional approach in testing RL software relies on *human oracles*, i.e. a panel of human experts are convened to manually judge the correctness of the RL software outputs (i.e. the agents). However, this approach heavily depends on the availability and quality (including experiences, subjective states, etc.) of the human experts, and cannot be fully automated.

When high quality human experts are unavailable, or when full automation is needed, we need an alternative to the human oracle. This is the focus of this chapter.

Specifically, we propose an oracle based on the Lyapunov stability control theory [79] [30]. The heuristics goes as follows.

It is well-known [79] [30] in control theory that given a linear system (including the linear state dynamics, and the linear controller), where the system's physical state at time t is described by an n-dimensional vector $X(t) \in \mathbb{R}^n$, then we can construct a so-called *Lyapunov equation*. If the equation has positive-definite symmetric matrix solution $\mathbf{P} \in \mathbb{R}^{n \times n}$, then the linear system is stable. Meanwhile, given this $\mathbf{P}$, we can define a so-called *Lyapunov potential energy* (aka *Lyapunov function*) $V(X(t)) \stackrel{\mathsf{def}}{=} X(t)^{\mathsf{T}}\mathbf{P}X(t)$, where $X(t)^{\mathsf{T}}$ is the transpose of $X(t)$. The Lyapunov potential energy $V(X(t))$ can only decrease over time.

We propose our RL software oracle by using the Lyapunov stability control theory inversely. We randomly generate many stable linear systems using the Lyapunov stability control theory. For each such system (including its linear state dynamics, its linear controller, and its Lyapunov potential energy $V(X(t))$), we input the following environment to the RL software under test: the state transition dynamics of the environment is the linear state dynamics. Meanwhile, we use $V(X(t))$ as the core of the environment's reward function, and punish any agent (i.e. the RL learnt controller) that ever increases $V(X(t))$ over time. Expectedly, if the RL software is bug-less, the RL learnt controller should be smooth (for example, linear), aware of the hidden Lyapunov potential energy $V(X(t))$ of the environment, and its control actions shall always decrease $V(X(t))$. Otherwise, the RL software is considered buggy.

Based on the above RL software oracle design heuristics, this paper makes the following contributions.

1. As an alternative to the traditional human oracle for testing RL software, we

propose a Lyapunov stability control theory based oracle, LPEA(ϑ, θ).

2. We conduct extensive experiments over representative RL algorithms and RL software bugs to evaluate our proposed oracle. The results show that under various configurations of (ϑ, θ), our LPEA(ϑ, θ) oracles can achieve good performances in most metrics. The average accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC are respectively 75.2%, 94.6%, 55.7%, 68.6%, 55.7%, 94.8%, 5.2%, 44.3%, and 0.751.

3. We conduct extensive experiments over representative RL algorithms and RL software bugs to compare our proposed oracle with the human oracle. The results show that our proposed oracle can outperform the human oracle in most metrics. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve's AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

The rest of the chapter is organized as follows. Section 4.2 provides background on RL and Lyapunov stability control theory. Section 4.3 proposes our oracle for testing RL software. Section 4.4 evaluates our proposal. Section 4.5 discusses related work. Section 4.6 concludes the chapter.

4.2 Background

4.2.1 Reinforcement Learning

Reinforcement Learning (RL) is a paradigm of machine learning, where the RL software trains an *agent* by letting it interact with an inputted environment. The key concepts related to RL include:

1. **Environment:** The external system with which the agent interacts. It includes the following concepts.

   (a) **State:** We denote the state at time t of the environment as $X(t)$. The set of all possible states is called the *state space*, denoted as $\mathcal{X}$.

   (b) **Action:** We denote the action applied to the environment at time t as $U(t)$. The set of all possible actions is called the *action space*, denoted as $\mathcal{U}$.

   (c) **State Transition Dynamics:** Given the current state $X(t)$ and action $U(t)$ of the environment, the way that the state $X(t)$ will evolve is called the environment's *state transition dynamics*.

   (d) **Reward Function:** A function $r : \mathcal{X} \times \mathcal{U} \mapsto \mathbb{R}$ that maps the current state $X(t)$ and action $U(t)$ to a scalar real value (aka the *reward*), which measures the desirability of the current state and action.

2. **Agent:** The learning entity that interacts (issue actions to and receive states/rewards from) with the environment. The RL software takes charge of structuring and modifying the agent, aiming to maximize the cumulative reward. The final version of the agent is the output of the RL software.

4.2.2 Lyapunov Stability Control Theory

The most well-studied control theories are about *linear systems* [20], where the state of a system at time t can be modeled by an n-dimensional vector $X(t) \in \mathbb{R}^n$, and it evolves according to the so called *linear state dynamics*:

$$X(t+1) = \mathbf{A}X(t) + \mathbf{B}U(t), \tag{4.1}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times m}$ are respectively n-by-n and n-by-m matrices; $U(t) \in \mathbb{R}^m$ is the m-dimensional action (i.e. controller feedback) applied to the system at time t; $X(t+1) \in \mathbb{R}^n$ is the state of the system at the next time tick, i.e. $(t+1)$; note as the focus is on testing RL software, which is discrete, we only use the discrete-time control theories in this chapter.

Besides the linear state dynamics, the other key component of a linear system is the *linear controller*, which takes the following form:

$$U(t) = -\mathbf{K}X(t), \tag{4.2}$$

where $\mathbf{K} \in \mathbb{R}^{m \times n}$ is a m-by-n matrix.

The key to designing the linear controller is to find a proper $\mathbf{K}$ for Eq. (4.2). Given $\mathbf{A}$ and $\mathbf{B}$ in Eq. (4.1), there are many well-established ways to propose candidate values for $\mathbf{K}$, e.g. LQR [12], LMI [18] etc.

Once a candidate value for $\mathbf{K}$ is proposed, we can construct the so-called *Lyapunov equation*:

$$\tilde{\mathbf{A}}^{\mathsf{T}}\mathbf{P}\tilde{\mathbf{A}} - \mathbf{P} + \mathbf{Q} = 0, \tag{4.3}$$

where $\tilde{\mathbf{A}} \stackrel{\mathsf{def}}{=} \mathbf{A} - \mathbf{B}\mathbf{K}$; $\tilde{\mathbf{A}}^{\mathsf{T}}$ is the transpose of $\tilde{\mathbf{A}}$; $\mathbf{Q} \in \mathbb{R}^{n \times n}$ is an arbitrarily given positive-definite symmetric matrix (e.g. we can set $\mathbf{Q}$ to be the identity matrix $\mathbf{I}$). The only unknown variable in Eq. (4.3) is $\mathbf{P} \in \mathbb{R}^{n \times n}$.

The well-known Lyapunov stability control theory [79] [30] says, if Eq. (4.3) has a positive-definite symmetric matrix solution $\mathbf{P} \in \mathbb{R}^{n \times n}$, then the original linear system

of Eq. (4.1)(4.2) is *globally asymptotically stable*. That is, starting from any initial state $X(0)$, the trajectory of $X(t)$ will always converge to the origin point (denoted as $\mathbf{0}$) of the state space $\mathbb{R}^n$. In this chapter, we simply refer to "globally asymptotically stable" as "*stable*."

There are well-established tools to solve the Lyapunov equation, i.e. to find a positive-definite symmetric matrix $\mathbf{P}$ that upholds Eq. (4.3) [18] [94]. When a solution cannot be found, there are also well-established tools to propose other candidate values for $\mathbf{K}$ (i.e. propose other linear controllers), so as to change the Lyapunov equation. But still, there are chances that for certain given $\mathbf{A}$ and $\mathbf{B}$, no proper $\mathbf{K}$ exists to construct a solvable Lyapunov equation. Fortunately, such chances are low in practice.

Once we solve the Lyapunov equation, i.e. found a positive-definite symmetric matrix $\mathbf{P} \in \mathbb{R}^{n \times n}$ that upholds Eq. (4.3), we can define the so-called *Lyapunov potential energy*, aka *Lyapunov function*:

$$V(X(t)) \stackrel{\mathsf{def}}{=} X(t)^{\mathsf{T}} \mathbf{P} X(t), \tag{4.4}$$

where $X(t)^{\mathsf{T}}$ is the transpose of $X(t)$. The Lyapunov stability control theory also tells us that the Lyapunov potential energy $V(X(t))$ always decreases over time.

4.3 Solution

Suppose a to-be-tested RL software implementation R takes in as input an environment $\mathcal{E}$, then trains and outputs an agent a. Following the heuristics described in Section 4.1, we propose our RL software oracle as follows.

- **Step 1:** Generate I stable[3] linear systems $\{\mathcal{S}_i\}_{i=1}^{I}$. System $\mathcal{S}_i$ ($i = 1, \ldots, I$) includes linear state dynamics of

$$X_i(t+1) = \mathbf{A}_i X_i(t) + \mathbf{B}_i U_i(t), \tag{4.5}$$

where $X_i(t) \in \mathbb{R}^n$, $U_i(t) \in \mathbb{R}^m$, $\mathbf{A}_i \in \mathbb{R}^{n \times n}$, $\mathbf{B}_i \in \mathbb{R}^{n \times m}$; and a linear controller of

$$U_i(t) = -\mathbf{K}_i X_i(t). \tag{4.6}$$

where $\mathbf{K}_i \in \mathbb{R}^{n \times m}$.

Meanwhile, as a stable linear system, $\mathcal{S}_i$ shall also have a Lyapunov potential energy (aka Lyapunov function) defined as

$$V_i(X_i(t)) \stackrel{\mathsf{def}}{=} X_i(t)^{\mathsf{T}} \mathbf{P}_i X_i(t), \tag{4.7}$$

where $X_i(t)^{\mathsf{T}}$ is the transpose of vector $X_i(t)$, and $\mathbf{P}_i \in \mathbb{R}^{n \times n}$ is an n-by-n positive-definite symmetric matrix that solves the Lyapunov equation of $\mathcal{S}_i$ (see Section 4.2.2 and Eq. (4.3)(4.4)).

- **Step 2:** For each linear system $\mathcal{S}_i$, we construct an environment $\mathcal{E}_i$ as follows. First, we adopt $X_i(t)$, $U_i(t)$, and Eq. (4.5) of $\mathcal{S}_i$ respectively as $\mathcal{E}_i$'s state, action, and state transition dynamics (see Section 4.2.1). Second, we define the reward function r_i of $\mathcal{E}_i$ as follows.

$$\begin{aligned} & r_i(X_i(t), U_i(t)) \\ \stackrel{\mathsf{def}}{=} & \ \mathsf{exp}(-V_i(X_i(t))) + \alpha\, \mathsf{exp}(-||X_i(t)||_2) \\ = & \ \mathsf{exp}(-X_i(t)^{\mathsf{T}} \mathbf{P}_i X_i(t)) + \alpha\, \mathsf{exp}(-||X_i(t)||_2), \end{aligned} \tag{4.8}$$

where $\alpha \in [0, 1]$ is a weighting constant, exp is the exponent function, and $||X||_2$ is the 2-norm of vector X (i.e. vector X's Euler length). That is, *we reward the*

[3]According to Section 4.2.2, given any arbitrarily generated $\mathbf{A}$ and $\mathbf{B}$ for Eq. (4.1), there are chances that a stable linear system cannot be designed. However, such chance is low in practice. Hence generating I stable linear systems is practically feasible, e.g. via repetitive trial.

decrease of the Lyapunov potential energy, and we reward converging $X_i(t)$ to the origin point $\mathbf{0}$ in $\mathbb{R}^n$.

Note we exclude $U_i(t)$ from the right hand side of Eq. (4.8) to feedback less info to the RL, so as to increase the learning difficulty, hoping to expose more problems of the RL software implementation.

Also note an agent (aka "controller" in this context) that matches such reward function exists, which is the linear controller described by Eq. (4.6). According to the Lyapunov stability control theory described in Section 4.2.2, as the linear system $\mathcal{S}_i$ is stable, the linear controller of Eq. (4.6) guarantees the Lyapunov potential energy always decreases over time, and $X(t)$ converges to $\mathbf{0}$. An ideal RL software shall figure out this controller based on the reward function of Eq. (4.8).

- **Step 3:** Input each environment $\mathcal{E}_i$ ($i = 1, \ldots, I$) to the to-be-tested RL software implementation R, which respectively outputs the trained agent (aka "controller" in this context) a_i. Given any state $X_i(t) \in \mathbb{R}^n$, controller a_i maps $X_i(t)$ to an action of $a_i(X_i(t)) \in \mathbb{R}^m$. In other words, a_i is a function of $\mathbb{R}^n \mapsto \mathbb{R}^m$. We then define a *controller goodness* function to quantify the quality of a_i with a scalar value, and denote this function as $g_i(a_i)$. Note $g_i(a_i) \in \mathbb{R}$. Upon $\{g_i(a_i)\}_{i=1}^{I}$, we can further define an *implementation goodness* function to quantify the quality of R. Denote the implementation goodness function as $q(\{g_i(a_i)\}_{i=1}^{I})$. Note $q(\{g_i(a_i)\}_{i=1}^{I}) \in \mathbb{R}$. We can then set a threshold constant $\theta \in \mathbb{R}$, and claim R to be "buggy" if $q(\{g_i(a_i)\}_{i=1}^{I}) < \theta$, and "bug-less" otherwise.

Step 1 $\sim$ 3 is not yet an oracle, unless the controller/implementation goodness functions and threshold θ are defined. We propose to define them as follows, and name the resulted oracle the *Lyapunov Potential Energy Abnormality* (LPEA) oracle.

For each controller a_i ($i = 1, \ldots, I$) outputted in **Step 3**, we uniformly random sample J initial states in the state space $\mathcal{X}_i = \mathbb{R}^n$, and denote these initial states as $\{X_{i,j}(0)\}_{j=1}^{J}$. Correspondingly, a_i generates the respective actions $\{a_i(X_{i,j}(0))\}_{j=1}^{J}$ to be fed to the environment $\mathcal{E}_i$. Then $\mathcal{E}_i$ will respectively evolve the J initial states by 1 time tick, to generate the next states $\{X_{i,j}(1)\}_{j=1}^{J}$. The corresponding change of Lyapunov potential energy is

$$\Delta V_{i,j}(0) \stackrel{\mathsf{def}}{=} V_i(X_{i,j}(1)) - V_i(X_{i,j}(0)). \tag{4.9}$$

Correspondingly, define

$$\mathcal{V}_i^- \stackrel{\mathsf{def}}{=} \{j | j \in \{1, \ldots, J\} \text{ and } \Delta V_{i,j}(0) < 0\}.$$

Apparently, $0 \leqslant |\mathcal{V}_i^-| \leqslant J$, hence $\frac{|\mathcal{V}_i^-|}{J} \in [0, 1]$.

We have the following conjecture.

Conjecture 1. *If the RL software implementation R is bug-less, then it can figure out the underlying meaning of the reward function Eq.* (4.8)*. Particularly, almost all of the outputted agent (controller) a_i ($i = 1, \ldots, I$) shall largely comply with the Lyapunov stability control theory: the Lyapunov potential energy shall decrease over time. That is, almost all $\Delta V_{i,j}(0)$s ($i = 1, \ldots, I$; $j = 1, \ldots, J$) should be negative; in other words, almost all $\frac{|\mathcal{V}_i^-|}{J}$s ($i = 1, \ldots, I$) should be close to* 100%*. Otherwise, R is buggy.*

Based on Conjecture 1, we define the following *controller goodness function*

$$g_i(a_i) \stackrel{\mathsf{def}}{=} \begin{cases} 1 & (\text{if } \frac{|\mathcal{V}_i^-|}{J} \geqslant \vartheta), \\ 0 & (\text{otherwise}), \end{cases} \tag{4.10}$$

where $\vartheta \in [0, 1]$ is a configurable threshold constant.

Furthermore, we should set ϑ close to 100%. In this way, according to Conjecture 1, if the RL software implementation R is bug-less, then almost all $g_i(a_i)$s ($i = 1, \ldots, I$) will take the high value of "1"; while if R is buggy, then many $g_i(a_i)$s will take the low value of "0". $(*)$

Based on the controller goodness function, we define the *implementation goodness function*

$$q(\{g_i(a_i)\}_{i=1}^{I}) \stackrel{\mathsf{def}}{=} \frac{\sum_{i=1}^{I} g_i(a_i)}{I}. \tag{4.11}$$

Apparently, $q(\{g_i(a_i)\}_{i=1}^{I}) \in [0, 1]$. Hence we can configure the threshold constant θ to a value in $[0, 1]$, and claim the RL software implementation R "bug-less" if $q \geqslant \theta$, and "buggy" if $q < \theta$.

Furthermore, due to $(*)$, if R is bug-less, then we can conjecture almost all $g_i(a_i)$s ($i = 1, \ldots, I$) will take the value of "1", raising the q value close to "1"; while if R is buggy, then we can conjecture many $g_i(a_i)$s ($i = 1, \ldots, I$) will take the value of "0", dropping the q value toward "0." $(**)$

We can see different configurations of the ϑ and θ values result in different LPEA oracles, denoted respectively as LPEA(ϑ, θ). Though $(*)$ and $(**)$ give guidelines on the configuration of the (ϑ, θ) value, the optimal configuration of the (ϑ, θ) value is still an open problem. A brute force solution is to exhaustively try a reasonable set of candidate values. For example, guided by $(*)$ and $(**)$, we can try $\vartheta = 100\%$, 97.5%, 95%, 92.5%, 90%, and $\theta = 75\%$, 50%, 25%, resulting in $5 \times 3 = 15$ different combinations: LPEA$(100\%, 75\%)$, LPEA$(100\%, 50\%)$, ..., LPEA$(90\%, 25\%)$.

4.4 Evaluation

4.4.1 Overall Evaluation Process, Metrics, and Research Questions

We are interested in comparing our proposed oracles with the traditional human oracle in testing RL software.

Specifically, we create benchmarks containing multiple implementations of RL software $\{R_{k,l}\}_{k=1,\ldots,K;l=0,1,\ldots,L_k}$, where $R_{k,0}$ ($k = 1, \ldots, K$) is the bug-less implementation of the kth RL software, and $R_{k,l}$ ($l = 1, \ldots, L_k$, where $L_k \in \mathbb{N}^+$) is the lth buggy implementation of the kth RL software. Define

$$\mathcal{R}' \stackrel{\text{def}}{=} \{R_{k,0} | k = 1, \ldots, K\}, \tag{4.12}$$

$$\bar{\mathcal{R}} \stackrel{\text{def}}{=} \{R_{k,l} | k = 1, \ldots, K \text{ and } l = 1, \ldots, L_k\}. \tag{4.13}$$

Note for each piece of RL software, if we adopt only one bug-less implementation but multiple buggy implementations, then $|\mathcal{R}'| << |\bar{\mathcal{R}}|$. This makes the benchmark imbalanced. To balance the benchmark, for each bug-less RL software implementation $R_{k,0}$ ($k = 1, \ldots, K$), we create $(L_k - 1)$ additional bug-less implementations: $\{R_{k,-l}\}_{l=1,\ldots,(L_k-1)}$, where $R_{k,-l}$ is a varied bug-less implementation of $R_{k,0}$ by adding random irrelevant instruction(s) into $R_{k,0}$, e.g., by adding no-op instructions randomly into $R_{k,0}$. Correspondingly, we define

$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{R}' \cup \{R_{k,-l} | k = 1, \ldots, K \text{ and } l = 1, \ldots, (L_k - 1)\}. \tag{4.14}$$

We call $\mathcal{R}'$ the *raw bug-less RL software benchmark*, while $\mathcal{R}$ the *expanded bug-less RL software benchmark* (or simply the *bug-less RL software benchmark*). We call $\bar{\mathcal{R}}$ the *buggy RL software benchmark*; and call $(\mathcal{R} \cup \bar{\mathcal{R}})$ the *RL software benchmark*.

Given oracle ω, we denote

$$\mathcal{R}_\omega^\ominus \stackrel{\text{def}}{=} \{R | R \in \mathcal{R} \text{ and is labeled "bug-less" by } \omega\}, \tag{4.15}$$

$$\mathcal{R}_\omega^\oplus \stackrel{\text{def}}{=} \{R | R \in \mathcal{R} \text{ and is labeled "buggy" by } \omega\}, \tag{4.16}$$

$$\bar{\mathcal{R}}_\omega^\ominus \stackrel{\text{def}}{=} \{R | R \in \bar{\mathcal{R}} \text{ and is labeled "bug-less" by } \omega\}, \tag{4.17}$$

$$\bar{\mathcal{R}}_\omega^\oplus \stackrel{\text{def}}{=} \{R | R \in \bar{\mathcal{R}} \text{ and is labeled "buggy" by } \omega\}. \tag{4.18}$$

We define the following key metrics for ω:

$$\text{True Positive Count TP}(\omega) \stackrel{\text{def}}{=} |\bar{\mathcal{R}}_\omega^\oplus|, \tag{4.19}$$

$$\text{False Positive Count FP}(\omega) \stackrel{\text{def}}{=} |\mathcal{R}_\omega^\oplus|, \tag{4.20}$$

$$\text{True Negative Count TN}(\omega) \stackrel{\text{def}}{=} |\mathcal{R}_\omega^\ominus|, \tag{4.21}$$

$$\text{False Negative Count FN}(\omega) \stackrel{\text{def}}{=} |\bar{\mathcal{R}}_\omega^\ominus|, \tag{4.22}$$

$$\text{Accuracy } \gamma_{\mathsf{AC}}(\omega) \stackrel{\text{def}}{=} \frac{|\mathcal{R}_\omega^\ominus| + |\bar{\mathcal{R}}_\omega^\oplus|}{|\mathcal{R}| + |\bar{\mathcal{R}}|}. \tag{4.23}$$

Additional metrics like precision, recall, and F1 score are standard and well-defined in the literature, so they are not explicitly redefined here to save space.

We want to study the following research question:

RQ: How good is the LPEA(ϑ, θ) oracle compared to the human oracle?

4.4.2 Bug-less RL Software Benchmark

To implement the overall evaluation process described in Section 4.4.1, we still need to address several problems.

The first implementation problem is which pieces of RL software shall be included in the bug-less RL software benchmark $\mathcal{R}$ (see Eq. (4.14)). Or more fundamentally, which pieces of RL software shall be included in the *raw* bug-less RL software benchmark $\mathcal{R}'$ (see Eq. (4.12)).

One choice is the *Stable Baselines3* (SB3) [75] [2] library, a reasonably sized (over 100 source code files, and over 18000 lines of source code (excluding comments and blanks)), well-known and well-maintained open-source implementation of existing RL algorithms. There are two main families of existing RL algorithms: *on-policy* and *off-policy* (see [87] for their rigorous definitions). One of the most representative on-policy RL algorithms is *Advantage Actor-Critic* (A2C) [61]; while one of the most representative off-policy RL algorithms is *Twin Delayed Deep Deterministic policy gradient* (TD3) [29], a descendant of the well-known Q-Learning algorithm [97]. We therefore include the state-of-the-art SB3 stable implementations of A2C and TD3 into our raw bug-less RL software benchmark $\mathcal{R}'$. Note this assumes the state-of-the-art SB3 stable implementations are trustworthy enough to be regarded as the *ground-truth* bug-less implementations. We have to adopt this best-effort practice because perfectly bug-less implementations are empirically unavailable.

In addition to A2C and TD3, OpenAI also strongly recommends the *Proximal Policy Optimization* (PPO) RL algorithm [55] [68]. Therefore, we also include the state-of-the-art SB3 stable implementation of PPO into $\mathcal{R}'$.

In summary, the bug-less RL software included in $\mathcal{R}'$ are the SB3 stable implementation (specifically, the GitHub commit hash is `4efee92`, see [75] [2]) of A2C, PPO, and TD3. Note both A2C and PPO are on-policy RL algorithms, and TD3 is an off-policy algorithm. Also note A2C, PPO, and TD3 are general purpose RL algorithms for both continuous and discrete state/action environments.

We then carry out harmless variations (see the paragraph before Eq. (4.14) in Section 4.4.1) upon the state-of-the-art SB3 stable implementations of A2C, PPO, and TD3, to expand $\mathcal{R}'$ to $\mathcal{R}$, i.e. creating the bug-less RL software benchmark $\mathcal{R}$. Specifically, $\mathcal{R}$ includes respectively 19, 22, and 15 bug-less implementations of A2C, PPO, and TD3. These numbers respectively match the numbers of buggy implementations of A2C, PPO, and TD3 included in $\bar{\mathcal{R}}$, the buggy RL software benchmark (see Section 4.4.3 paragraph $(***)$). In this way, the bug-less and buggy benchmarks, $\mathcal{R}$ and

$\bar{\mathcal{R}}$, are balanced.

4.4.3 Buggy RL Software Benchmark

The second implementation problem is how to create the *buggy* RL software benchmark $\bar{\mathcal{R}}$ (see Eq. (4.13)).

We create $\bar{\mathcal{R}}$ by injecting bugs into the bug-less RL software implementations included in $\mathcal{R}'$. Specifically, we inject two kinds of bugs: *artificial* and *real-world*.

We create the artificial bugs based on state-of-the-art surveys of RL software bugs. Specifically, as per the survey of Nikanjam et al. [66], Table 4.1 summarizes the types of RL software bugs[4].

Table 4.1: RL Software Bug Types from Survey [66]

Level-1 Type	Level-2 Type
1. RL Specific	1.1. Exploring the environment bugs
	1.2. Updating network bugs
2. NN Specific	2.1. Model bugs
	2.2. Training bugs
	2.3. Tensor and inputs bugs

Based on Table 4.1, we injected artificial bugs into the bug-less A2C, PPO, and TD3 implementations of $\mathcal{R}'$, to create various buggy implementations, collectively referred to as $\bar{\mathcal{R}}_{\mathsf{art}}$. Respectively, there are 17, 21, and 15 buggy implementations of A2C, PPO, and TD3 in $\bar{\mathcal{R}}_{\mathsf{art}}$. Table 4.2 summarizes the distributions of various types of

[4]As mentioned in footnote 2, this chapter focuses on hidden bugs, hence the non-hidden bugs (those show themselves via compilation warnings/failures and/or runtime exceptions/failures) listed in [66] are not listed in Table 4.1.

bugs in these implementations. Note the distribution is not even. This is because whether we can inject a specific type of bug into a bug-less implementation depends on the semantics of the bug and the implementation. Some semantics are easy for injection, some are difficult, even impossible.

Table 4.2: Number of Buggy Implementations in $\bar{\mathcal{R}}_{\mathsf{art}}$

Artificial Bug's Type*	A2C	PPO	TD3
1.1	0	0	1
1.2	13	16	8
2.1	1	1	1
2.2	2	2	4
2.3	1	2	1

* See Table 4.1 column 2 for the meanings of the bug type IDs.

Meanwhile, we collect real-world bugs by manually analyzing the GitHub commit history of SB3 [75] [2], including SB3's ancestor repository: *Stable Baselines* (SB) [38]. Many real-world bugs recorded in the commit history are *non-hidden bugs* (i.e. they show themselves via compilation warnings/failures and/or runtime exceptions/failures), which can be detected without oracle, hence are not the focus of this chapter. Our manual analysis finally collected 2 real-world hidden bugs, listed in Table 4.3. Bug RW1 resides in a parent class used by both the A2C and PPO implementations. By re-injecting bug RW1 into the bug-less A2C and PPO implementations respectively, we create two buggy implementations, respectively for A2C and PPO. Bug RW2 resides in a class used by the A2C implementation. By re-injecting bug RW2 into the bug-less A2C implementation, we create another buggy implementation of A2C. We refer to the three created buggy implementations collectively as $\bar{\mathcal{R}}_{\mathsf{rw}}$, and thus $\bar{\mathcal{R}} = \bar{\mathcal{R}}_{\mathsf{art}} \cup \bar{\mathcal{R}}_{\mathsf{rw}}$.

Based on the above narrations, we can see $\bar{\mathcal{R}}_{\mathsf{art}}$ includes respectively 17, 21, and 15 buggy (using artificial bugs) implementations of A2C, PPO, and TD3; while $\bar{\mathcal{R}}_{\mathsf{rw}}$

Table 4.3: Real-World Bugs

Bug ID	Description
RW1	SB3 GitHub commit hash: `d5986da` File: `stable_baselines3/common/on_policy_algorithm.py` Line 167: "`dones`" should be "`_last_dones`"; Line 178: should insert one more line "`self._last_dones = dones`" before this line.
RW2	SB GitHub commit hash: `689afd1` File: `stable_baselines/a2c/a2c.py` Line 325: "`np.int32`" should be "`self.env.action_space.dtype`".

includes respectively 2, 1, and 0 buggy (using real-world bugs) implementations of A2C, PPO, and TD3. Therefore, $\bar{\mathcal{R}}$ includes respectively $(17+2) = 19$, $(21+1) = 22$, and $(15+0) = 15$ buggy implementations of A2C, PPO, and TD3. $(***)$

4.4.4 Human Oracle and Comparison Configurations

The third implementation problem is the design of human oracle, the traditional oracle for testing RL software.

However, so far, there is no standard specification on the design of human oracle for testing RL software. To our best knowledge, the most relevant specification is given by Zolfagharian et al. through their paper on STARLA [106]. The focus of STARLA is on how to use genetic algorithms to create test cases for RL software. Nevertheless, it also mentions how to judge if an agent (the output of RL software) is abnormal: by checking if the agent drives the test environment into a "*faulty state*;" and the definitions of "faulty states" are given by human experts.

However, the STARLA specification still lacks details (after all, STARLA's focus is not on oracle). In the following, we expand the STARLA specification into a more detailed human oracle specification, assuming the to-be-tested RL software implementation is R.

- **Step 1:** We choose I classic environments $\{\mathcal{E}_i\}_{i=1}^{I}$.
- **Step 2:** Repeat **Step 2.1** H ($H \in \mathbb{N}^+$) times.

  Step 2.1: In the hth ($h \in \{1, \ldots, H\}$) iteration, input each environment $\mathcal{E}_i$ ($i = 1, \ldots, I$) to R, which outputs the corresponding trained agent $a_{i,h}$ ($i = 1, \ldots, I$). Let $a_{i,h}$ interact with $\mathcal{E}_i$, and record the *interaction trace*, which is a time series of the states of $\mathcal{E}_i$, denoted as $\{\mathsf{state}_{i,h,t}\}_t$.
- **Step 3:** Distribute the recorded IH interaction traces to human experts.
- **Step 4:** For each human expert, and for each interaction trace she/he got, she/he *subjectively*[5] decides a set of "*faulty states*." The interaction trace is labeled "*abnormal*" if the trace ever reaches a "faulty state;" and is labeled "*normal*" otherwise.
- **Step 5:** R is labeled "*buggy*" if there exists an "abnormal" interaction trace; and is labeled "*bug-less*" otherwise.

In practice, we choose the classic *Mountain Car Continuous* (MCC) and the *Pendulum* environments from the famous Gymnasium repository [90] [28], respectively as our $\mathcal{E}_1$ and $\mathcal{E}_2$ (hence $I = 2$). We choose $H = 3$.

We use the same bug-less RL software benchmark $\mathcal{R}$ and buggy RL software benchmark $\bar{\mathcal{R}}$ respectively described in Section 4.4.2 and 4.4.3. As mentioned in Section 4.4.2, $\mathcal{R}$ includes respectively 19, 22, and 15 bug-less implementations of A2C,

[5]The human experts have the freedom to check all the common statistical metrics on the provided traces. In this sense, the common statistical metrics are already included in the human oracle design.

PPO, and TD3 RL software. As mentioned in Section 4.4.3, $\bar{\mathcal{R}}$ includes respectively 19, 22, and 15 buggy implementations of A2C, PPO, and TD3 RL software. Hence in total, we have $|\mathcal{R}| + |\bar{\mathcal{R}}| = (19 + 22 + 15) + (19 + 22 + 15) = 112$ different RL software implementations. Together these implementations create $(|\mathcal{R}| + |\bar{\mathcal{R}}|)IH = 112 \times 2 \times 3 = 672$ interaction traces.

We randomly distribute the 672 interaction traces to a panel of three human experts. Two of the human experts have bachelor's degrees in computer science, and the other has PhD degree in computer science and robotics related engineering. The three panel members respectively reported spending at least 1.5, 2, and 2.5 hours on labeling these traces. This totals at least 6 human-hour of human resources allocated to run the human oracle.

In contrast, the human resources needed to run our proposed LPEA(ϑ, θ) oracle is always nearly 0 human-hour (no matter what ϑ and θ values we choose): the human operator only needs to enter a command line to start the automated workflow. Our evaluations hence are biased in favour of the human oracle in the sense that way more human resources are allocated to it than to the LPEA(ϑ, θ) oracle.

In terms of computing resources. The human oracle costs 24 hours of computing, mainly to generate the $(|\mathcal{R}| + |\bar{\mathcal{R}}|)IH = 672$ interaction traces. The computing platform is a laptop with AMD Ryzen 7 5800H CPU and NVIDIA GeForce RTX 3060 Laptop GPU.

For an LPEA(ϑ, θ) oracle (see Section 4.3), we choose the following learning hyper-parameters: $I = 20$; $J = 800$; $n = 2$; $m = 2$; learning rate for A2C, PPO, and TD3 are respectively 0.0004, 0.0012, and 0.001; learning steps for A2C, PPO, and TD3 are respectively 90000, 120000, and 90000. The resulted computing cost is always within 129 hours (no matter what ϑ and θ values we choose), using the same computing platform as that for the human oracle.

As the computing platform costs USD1600 to purchase, and can be used for at least

3 years. This corresponds to an hourly price of 0.061USD/hour. In contrast, the human resource cost is about 10USD/hour. Therefore, the total monetary cost of the human oracle is $10 \times 6 + 0.061 \times 24 = 61.464$(USD); while the total monetary cost of an LPEA oracle is $10 \times 0 + 0.061 \times 129 = 7.869$(USD). Therefore, in terms of total monetary cost, our evaluations are still biased in favour of the human oracle (i.e. more money is allocated to the human oracle).

4.4.5 Evaluation Results

Table 4.4: Raw Data

Oracle	A2C				PPO				TD3				Overall (A2C, PPO, and TD3)			
	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
LPEA($\vartheta = 100\%, \theta = 75\%$)	15	0	19	4	15	0	22	7	15	15	0	0	45	15	41	11
LPEA($\vartheta = 100\%, \theta = 50\%$)	15	0	19	4	12	0	22	10	15	15	0	0	42	15	41	14
LPEA($\vartheta = 100\%, \theta = 25\%$)	13	0	19	6	9	0	22	13	14	12	3	1	36	12	44	20
LPEA($\vartheta = 97.5\%, \theta = 75\%$)	15	0	19	4	9	0	22	13	12	2	13	3	36	2	54	20
LPEA($\vartheta = 97.5\%, \theta = 50\%$)	15	0	19	4	8	0	22	14	10	0	15	5	33	0	56	23
LPEA($\vartheta = 97.5\%, \theta = 25\%$)	13	0	19	6	6	0	22	16	10	0	15	5	29	0	56	27
LPEA($\vartheta = 95\%, \theta = 75\%$)	15	0	19	4	8	0	22	14	11	0	15	4	34	0	56	22
LPEA($\vartheta = 95\%, \theta = 50\%$)	13	0	19	6	7	0	22	15	10	0	15	5	30	0	56	26
LPEA($\vartheta = 95\%, \theta = 25\%$)	11	0	19	8	5	0	22	17	8	0	15	7	24	0	56	32
LPEA($\vartheta = 92.5\%, \theta = 75\%$)	13	0	19	6	8	0	22	14	11	0	15	4	32	0	56	24
LPEA($\vartheta = 92.5\%, \theta = 50\%$)	13	0	19	6	6	0	22	16	10	0	15	5	29	0	56	27
LPEA($\vartheta = 92.5\%, \theta = 25\%$)	9	0	19	10	5	0	22	17	7	0	15	8	21	0	56	35
LPEA($\vartheta = 90\%, \theta = 75\%$)	13	0	19	6	8	0	22	14	10	0	15	5	31	0	56	25
LPEA($\vartheta = 90\%, \theta = 50\%$)	13	0	19	6	5	0	22	17	8	0	15	7	26	0	56	30
LPEA($\vartheta = 90\%, \theta = 25\%$)	8	0	19	11	5	0	22	17	7	0	15	8	20	0	56	36
Human Oracle	12	15	4	7	15	15	7	7	11	8	7	4	38	38	18	18

Table 4.4 lists all the raw data on the true positive count (TP), false positive count (FP), true negative count (TN), and false negative count (FN) (see Eq. (4.19)-(4.22)) over all oracles, including 15 LPEA oracles of different (ϑ, θ) configurations and the human oracle; while Fig. 4.1, Fig. 4.2, Fig. 4.3 and Fig. 4.5 plot the derived performance metrics.

Fig 4.1 visualizes the accuracy (see Eq. (4.23)), precision, recall, and F1 score metrics

using the overall data from Table 4.4. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values.

In terms of accuracy, the LPEA oracles show strong performance across different (ϑ, θ) configurations. The highest accuracy of 0.804 is achieved by LPEA($\vartheta = 97.5\%, \theta = 75\%$) and LPEA($\vartheta = 95\%, \theta = 75\%$). Even the lowest accuracy (0.679 by LPEA($\vartheta = 90\%, \theta = 25\%$)) is higher than the human oracle's accuracy of 0.5.

In terms of precision, many LPEA oracles achieve the perfect score of 1.0. These include LPEA($\vartheta = 97.5\%, \theta = 50\%$), LPEA($\vartheta = 97.5\%, \theta = 25\%$), and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the lowest precision (0.737 by LPEA($\vartheta = 100\%, \theta = 50\%$)) is higher than the human oracle's precision of 0.5.

In terms of recall, the human oracle, with a recall of 0.679, beats many of the LPEA oracles. But still, LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA(($\vartheta = 100\%, \theta = 50\%$) beat the human oracle respectively with a recall of 0.804 and 0.750. The high recall achieved by the human oracle is mainly due to the overly pessimistic (on whether an implementation is buggy) subjective views of the human experts. From Table 4.4-Overall-TP and FP columns, we can see that the human experts labeled (38+38) = 76 implementations out of the 112 implementations as "positive" (i.e. "buggy"), though only half of the implementations (i.e. 112/2 = 56) are actually buggy. These overly pessimistic subjective views lead to picking out most buggy implementations at the cost of mistakenly labeling many bug-less implementations as "buggy." That is, at the cost of other metrics, such as accuracy, precision, and F1 score.

In terms of F1 score, a metric that comprehensively combines the accuracy, precision, and recall metrics, nearly all the LPEA oracles (with F1 scores ranging from 0.600 to 0.776) outperform the human oracle (with an F1 score of 0.576), except LPEA($\vartheta = 92.5\%, \theta = 25\%$) and LPEA($\vartheta = 90\%, \theta = 25\%$) (respectively with a F1 score of 0.545 and 0.526). Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) achieves the highest F1 score, 0.776. Due to the comprehensiveness of F1 score, this evidences

that LPEA($\vartheta = 100\%, \theta = 75\%$) has the overall best performance compared to other LPEA configurations and the human oracle.

Fig 4.2 visualizes the false positive rate and false negative rate metrics using the overall data from Table 4.4. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values. For false positive rates and false negative rates, the lower the values, the better.

In terms of false positive rates, many LPEA oracles achieve the perfect score of 0.0. These include LPEA($\vartheta = 97.5\%, \theta = 50\%$), LPEA($\vartheta = 97.5\%, \theta = 25\%$), and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the highest false positive rate (0.268 by LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$)) is much lower than the human oracle's false positive rate of 0.679.

In terms of false negative rates, the human oracle, with a false negative rate of 0.321 beats many of the LPEA oracles. But still, LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$) beat the human oracle respectively with a false negative rate of 0.196 and 0.25. The reason for human oracle's good performance on false negative rate is similar to that on recall.

Fig 4.3 visualizes the true positive rate and true negative rate metrics using the overall data from Table 4.4. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values. For true positive rates and true negative rates, the higher the values, the better.

In terms of true positive rates, the human oracle, with a true positive rate of 0.679, beats many of the LPEA oracles. But still, LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$) beat the human oracle respectively with a true positive rate of 0.804 and 0.750. The reason for human oracle's good performance on true positive rate is similar to that on false negative rate (in fact, true positive rate and false negative rate always sum up to 1).

In terms of true negative rates, many LPEA oracles achieve the perfect score of 1.0.

These include LPEA($\vartheta = 97.5\%, \theta = 50\%$), LPEA($\vartheta = 97.5\%, \theta = 25\%$), and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the lowest true negative rate (0.732 by LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$)) is much higher than the human oracle's true negative rate of 0.321.

In addition to the metrics visualized by Fig. 4.4, Fig. 4.5 visualizes the ROC curves [60] using the raw data from Table 4.4. In the figure, the X-axis is the false positive rate and the Y-axis is the true positive rate. Similar to the F1 score, ROC curve provides another way to comprehensively reflect a classification tool's performance. An ideal ROC curve should be bent toward the top-left corner of the plot as much as possible, reaching high true positive rates at the cost of low false positive rates. Conversely, an ROC curve close to the diagonal line (aka the *random baseline*) represents a performance similar to random guessing, which should be avoided. Based on this intuition, a quantitative metric on ROC curve's quality is the *Area Under Curve* (AUC), and the bigger the AUC the better.

As per Fig. 4.5, in terms of the AUC, nearly all LPEA oracles are better than the human oracle, no matter based on the A2C, PPO, TD3, or overall data.

With all the data and metrics presented above, in summary and to answer the research question of **RQ** in Section 4.4.1, our LPEA(ϑ, θ) oracles (where ϑ = 100%, 97.5%, 95%, 92.5%, 90%, and θ = 75%, 50%, 25%) outperform the human oracle in majority of the metrics. The average accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC are respectively 75.2%, 94.6%, 55.7%, 68.6%, 55.7%, 94.8%, 5.2%, 44.3%, 0.751. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

4.4.6 Threats to Validity

1. Construct Validity Threats

1.1 Human Oracle Bias

The human experts serving as the human oracles can be biased due to their knowledge and skill differences. To make our evaluations more convincing, two of our three human experts have bachelor's degrees in computer science, and the other has PhD degree in computer science and robotics related engineering. This ensures they have reasonable knowledge and skill on computer science and are reasonably qualified to judge the behaviors of a piece of software. Though the qualities of the human experts can always be better, the human experts we recruit for this paper are reasonable compared to common practices.

The human experts' qualities are also validated by Fig 4.5. According to the figure, the AUC (Area Under the Curve, the bigger the better) of the human oracle (HO) ROC curves are significantly greater than that of the "random guessing" ROC curves (i.e. the diagonal dashed lines).

1.2 Ground Truth Validity

It is well-known that for any reasonably sized software (SB3 contains over 100 source code files and over 18000 lines of source code (excluding comments and blanks)), due to the explosion of combinatorial complexity, completely removing all bugs is empirically impossible. Therefore, the "bug-less" RL software implementations used in our evaluations can only be "bug-less" in the sense of best-effort. Specifically, we assume the state-of-the-art stable implementations of the well-known and well-maintained open-source SB3 library as trustworthy enough to be regarded as "bug-less."

1.3 Other Oracle Design Heuristics

There can be other heuristics to construct the oracle for testing RL software. We will

explore them in our future work. That said, one reason that the LPEA oracle attracts our attention is that the reward function requests a continuous and differentiable linear controller, hence the resulted changes of environment states form a continuous and differentiable vector field over the vector space of $\mathbb{R}^n$. A hidden bug usually triggers discrete fractures in such continuous and differentiable vector field, hence is easy to spot.

2. Internal Validity Threats:

2.1 Evaluation Platform Implementation Correctness

There can be bugs in the implementation of our oracles. To alleviate this threat, we conducted code reviews on the oracle implementation before the evaluations.

2.2 Bug Semantics and Injection Locations

The semantics and injection locations of the bugs are constrained by the bug types and the semantics of the bug-less implementations. On the other hand, due to combinatorial explosion of the semantics, we cannot try every possible bug semantics and injection location; but we are trying our best to randomize these bug semantics and injection locations.

3. External Validity Threats:

3.1 Human Oracle Representativeness

To our best knowledge, Zolfagharian et al.'s STARLA [106] paper provides the most rigorous definition in the literature on human oracle for testing RL software. The use of the Gymnasium repository [90] [28] is also a widely-adopted practice to evaluate the RL software implementations.

3.2 RL Software Implementations' Representativeness

We can always expand our benchmark to include more RL software implementations. However, we claim the SB3 implementations of A2C, PPO, and TD3 are reasonably

representative, because 1) they are representative algorithms covering the two main categories of RL algorithms: on-policy and off-policy [87]; 2) PPO is strongly recommended by OpenAI [55] [68]; and 3) SB3 is a reasonably sized, well-known and well-maintained open-source library of RL software implementations [75] [2].

4.5 Related Work

Metamorphic testing [22] has been explored to design oracles for testing machine learning software. Its main idea is to exploit domain-specific knowledge on the relations between the software inputs and outputs, and use such relations as oracles to detect erroneous software outputs. In this sense, the LPEA oracle that we propose in this chapter is a metamorphic testing oracle. It exploits the domain-specific knowledge on the decrease of Lyapunov potential energy.

Xie et al. [100] propose a metamorphic testing oracle for testing supervised machine learning software; and the follow up work of [101] proposes a metamorphic testing oracle for testing unsupervised machine learning software for clustering. However, this chapter focuses on testing reinforcement learning (RL) software, and RL is neither supervised learning nor unsupervised learning [87].

Besides metamorphic testing oracle, Pang et al. [70] and Tappler et al. [88] study how to test RL software for Markov decision processes. Both papers regard the controlled plant reaching an obvious faulty state (e.g. the plant is crashed) as the indicator (i.e. oracle) of buggy software. This concurs with our human oracle design (see Section 4.4.4 **Step 4**).

Padgham et al. [69] propose a model-based oracle generation method for automated unit testing of agents. This method needs white-box access to the design and implementation of the agents. Nikanjam et al. [66] propose a taxonomy of faults in RL software and propose DRLinter, a model-based fault detection framework for RL soft-

ware. DRLinter carries out static analysis and graph transformations upon the RL software, hence also needs white-box access to the RL software. Varshosaz et al. [92] propose ways to formally specify temporal-difference based RL algorithms, and check the behavioral properties on the specific steps of the algorithms as oracles. Obviously, this also needs white-box access to the RL software. In contrast, our proposed oracle only needs black-box access to the RL software and the agent. As future work, we can also compare with these white-box solutions. But in case the RL software or the agent source codes are unavailable/proprietary, we can then only use the black-box solutions.

The proposed LEPA oracle can be integrated into the frameworks of property-based testing, such as QuickCheck [24]. A property-based testing framework needs an oracle that exploits certain properties of the software output for testing. Our proposed LPEA oracle fits this need by exploiting the Lyapunov potential energy decreasing properties of the RL learned (outputted) agents.

Jothimurugan et al. [46] propose a formal language to specify RL problems and solutions. This work is also orthogonal to our LPEA oracle, which can be specified using this formal language in the future.

Shen et al. [83] and Hu et al. [41] propose to use mutation analysis [71] to evaluate the quality of a test set for machine learning software. Evaluating the quality of the test set (i.e. inputs to the learned model) is orthogonal to the oracle design, which focuses on labeling the outputs' correctness of the learned model. In addition, mutation analysis requires white-box access to the learned model; while our proposed oracle requires only black-box access of the learned model (agent).

Wan et al. [95] propose to use coverage-guided fuzzing to create better test sets for RL software. Again, the focus is on improving the quality of the test set, i.e. inputs to the learned model. While oracle focuses on the outputs of the learned model (agent). Hence the two problems are orthogonal to each other.

Finally, when we discuss the fairness of the evaluations, we compared the monetary costs of the proposed LPEA oracle and the human oracle (see Section 4.4.4). Another cost metric of interest is the environmental cost. Recent work by Lacoste et al. [53] quantified the carbon emissions of machine learning systems. Following this work, we can further measure the environmental cost of our proposed LPEA oracle. However, how to measure the environmental cost of human oracle is still an open problem.

4.6 Chapter Summary

In this Chapter, we address the RL software oracle problem by exploiting the Lyapunov stability control theory, and propose the LPEA(ϑ, θ) oracles. Our evaluations over representative RL algorithms and RL software bugs show that our LPEA(ϑ, θ) oracles (where ϑ = 100%, 97.5%, 95%, 92.5%, 90%, and θ = 75%, 50%, 25%) outperform the human oracle in most of the metrics. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

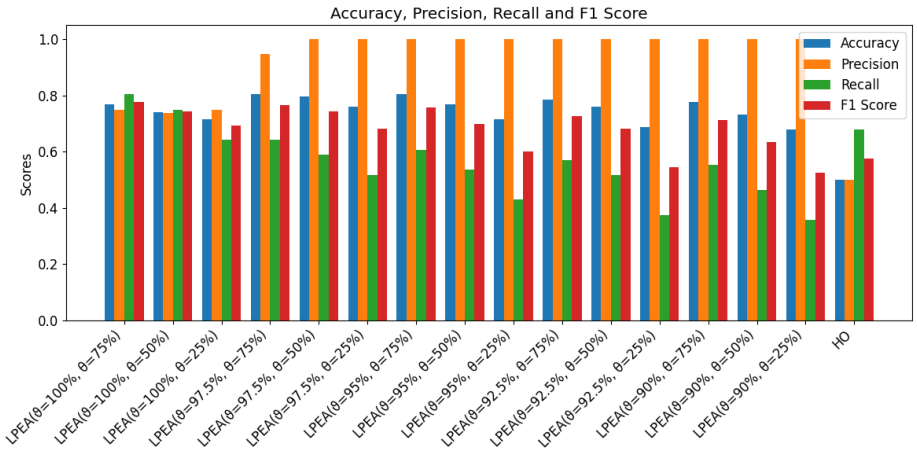


Figure 4.1: Accuracy, Precision, Recall, and F1 Score; higher values are better. HO: Human Oracle.

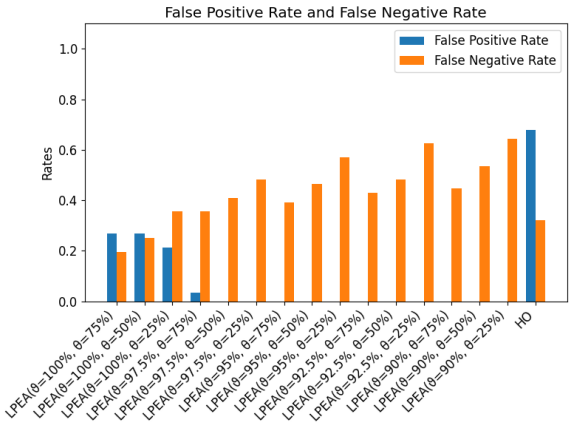


Figure 4.2: False Positive Rate and False Negative Rate; lower values are better. HO: Human Oracle.

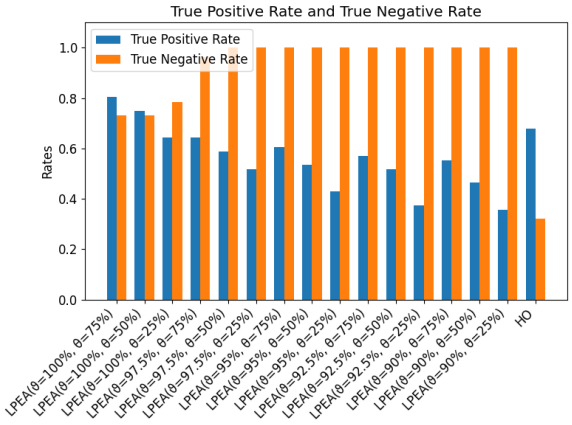
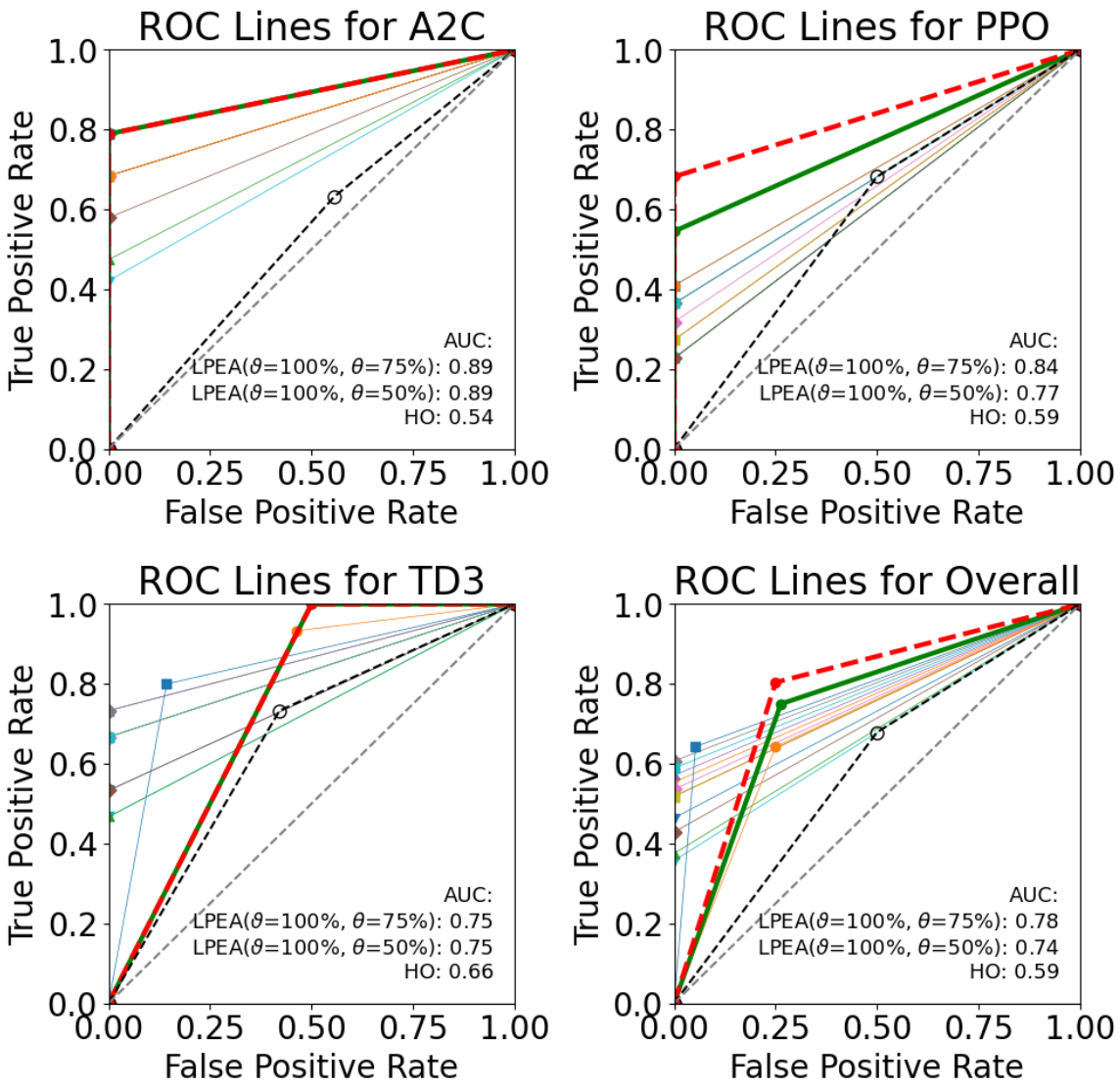


Figure 4.3: True Positive Rate and True Negative Rate; higher values are better. HO: Human Oracle

Figure 4.4: Performance of the LPEA Oracles and the *Human Oracle* (HO)



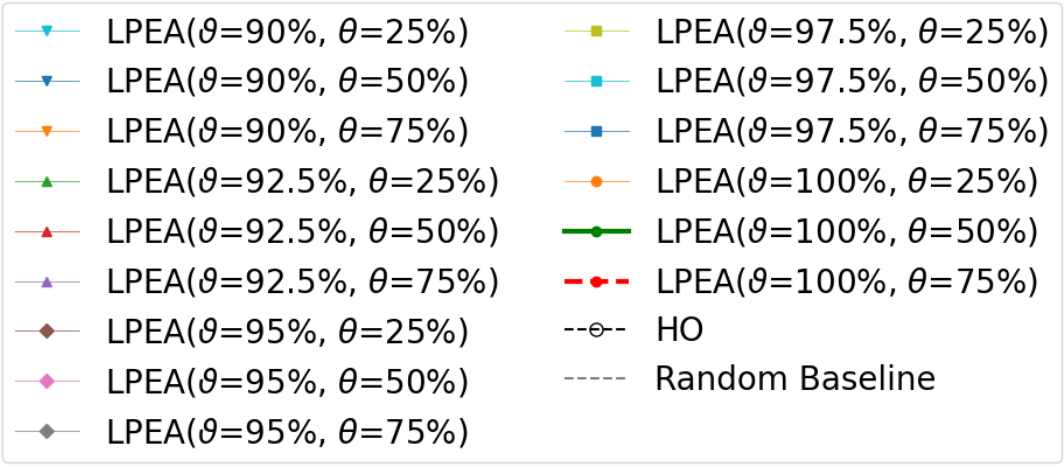


Figure 4.5: ROC Curves of the LPEA Oracles and the *Human Oracle* (HO)

Chapter 5

Discussions, Conclusions, and Future Work

5.1 Discussions

This section discusses key questions raised regarding the proposed oracles, focusing on three core dimensions: novelty and significance, parameter sensitivity, and limitations with applicable scenarios.

5.1.1 Novelty and Significance

- **Transformer-based Oracle (TO)**: Developed amid the rapid advancement of Transformer models, TO innovatively applies attention mechanisms to capture complex temporal dependencies in control-CPS trajectory data. While TO and AO demonstrate comparable overall performance in software fault localization quality, TO's lower false negative rate fills a critical gap for high-recall scenarios (e.g., safety-critical systems where missing bugs is intolerable), complementing AO's strength of a lower false positive rate.

- **Fuzzy Logic-guided Oracle**: Positioned as a problem-driven pilot study, this oracle explores fuzzy logic (a mature tool for handling ambiguity) to quantify RL policy compliance without relying on human experts—addressing the unique challenge of non-unique legitimate controllers and trajectories in RL. Though it does not outperform human oracles in most metrics, it provides insights for the development of the LPEA oracle.

- **LPEA Oracle**: Innovatively integrating Lyapunov stability control theory into RL oracle design. Unlike subjective human oracles, it uses the objective criterion of Lyapunov potential energy decrease to verify RL software validity. Evaluations show it outperforms human oracles in most key metrics, with the optimal configuration (LPEA($\vartheta = 100\%, \theta = 75\%$)) achieving 53.6% higher accuracy, 50% higher precision, and 127.8% higher true negative rate—validating the value of control theory integration for RL verification.

5.1.2 Parameter Sensitivity

- **Transformer-based Oracle (TO)**: TO's performance is sensitive to key hyper-parameters and environmental parameters. Hyper-parameters such as Transformer layer number and sliding window size p directly influence its ability to capture temporal dependencies. Environmental parameters like trajectory smoothness and noise level also impact performance: high-noise environments require more training data to filter interference.

- **Fuzzy Logic-guided Oracle**: Its performance is sensitive to task parameters and environment-dependent configurations. Task parameters including the number of training epochs E and state-action step count J affect the reliability of convergence trend analysis—insufficient E or J may lead to misjudgment of policy compliance. Environment-dependent configurations (e.g., fuzzy membership functions for discrete vs. continuous states) also matter: the oracle's

performance degrades in discrete environments (e.g., Frozen Lake vs. MountainCar) due to different compliance quantification (Section 3.4.5).

- **LPEA Oracle**: The number of randomly generated stable linear systems (I) is a critical sensitive parameter. Based on the conjecture that bug-less RL software consistently learns Lyapunov energy-decreasing policies across diverse stable linear environments, $I = 20$ was selected as a balance between statistical significance and computational feasibility (Section 4.4.4). Experimental results validate that $I = 20$ sufficiently distinguishes buggy from bug-less implementations, with generalization to A2C, PPO, TD3. Computationally, this configuration requires 129 hours (106 subjects $\times$ 20 control systems $\times$ 3.65 mins/training) on standard resources—further increasing I introduces redundant overhead without meaningful accuracy gains. Configuration parameters ϑ (controller goodness threshold) and θ (implementation goodness threshold) has been discussed in Section 4.4.5.

5.1.3 Limitations and Applicable Scenarios

- **Transformer-based Oracle (TO)**: TO is well-suited for slow-evolving control-CPS domains (e.g., UAVs, medical robots) where system behavior changes gradually. In such scenarios, its strengths in low false negative rates can be fully leveraged, while its reliance on stable training baselines is less problematic. A key limitation is its poor adaptability to rapidly changing systems : abrupt behavioral shifts make learned trajectory patterns obsolete, and parameter sensitivity is amplified in such environments.

- **Fuzzy Logic-guided Oracle and LPEA Oracle**: The core limitation is that it verifies RL software's fundamental ability to interact with environments and learn hidden policies, not its performance in specific complex application scenarios. It is well-suited for general RL software validity testing like detecting hidden

bugs in RL algorithm implementations, but faces challenges in high-dimensional nonlinear or dynamic open-world environments. For such specialized scenarios, further research is needed to adapt LPEA to domain-specific constraints, integrate scenario-aware metrics, and validate practical trade-offs.

5.2 Conclusions

This dissertation confronts the challenge of test oracle design for *Control Cyber-Physical Systems* (control-CPSs), known as the oracle problem. This challenge arises because conventional control-CPSs allow non-unique legitimate outputs—multiple valid physical trajectories—for a single control input. The problem is more severe in *Reinforcement Learning* (RL) based control-CPSs where both legitimate trajectories and controllers are non-unique. To address this, we develop and evaluate three novel test oracles that advance the state of the art.

For general control-CPS software, we design a *Transformer-based Oracle* (TO) and conducted systematic comparisons with the established *Auto-Regressive System Identification (AR-SI) Oracle* (AO). Our experimental evaluation reveals that both approaches demonstrate comparable overall performance in software fault localization quality, with no statistically significant differences in accuracy or latency. However, we observe important distinctions: AO has a lower false positive rate, while TO has a lower false negative rate.

For RL-based control-CPSs, we propose two specialized test oracles. The first is a fuzzy logic-guided oracle that quantifies behavioral compliance with reward policies. This oracle shows limited advantages over human oracles. While it outperformed human judgment in specific scenarios, most metrics reveal no significant improvement. In contrast, our *Lyapunov Potential Energy Abnormality* (LPEA)-based oracle demonstrates substantial superiority over human oracles across all evaluation metrics.

The most effective configuration, LPEA with parameters $\vartheta = 100\%$ and $\theta = 75\%$, improves accuracy by 53.6%, precision by 50%, true negative rate by 127.8%, and F1 score by 34.8%. Even the alternative LPEA configuration ($\vartheta = 100\%, \theta = 50\%$) delivers impressive gains, including 48.2% higher accuracy and 127.8% better true negative rate. These results validate the potential of leveraging control theory for RL verification.

Collectively, these contributions provide rigorous solutions to the control-CPS oracle problem, enabling more reliable automated testing for control-CPS.

5.3 Future Work

Built upon the work of this thesis, I plan to further explore the following topics.

Introduce more cross-domain oracle designs. Future efforts should explore the adaptation and integration of more control theories into the control-CPS oracle designs. This integration could yield more versatile oracles capable of handling broader scenarios.

Bridge the sim-to-real gap for digital-twin-trained controllers. Given that many control-CPS controllers are trained in simulated environments, their behavioral fidelity in real-world remains uncertain. To address this problem, I shall develop oracles to detect subtle performance degradations caused by the sim-to-real gaps.

References

[1] Ardupilot open source autopilot. Available at: https://github.com/ArduPilot/ardupilot.

[2] Github stable baselines3 repository. (Date last accessed 2-Aug-2024).

[3] Box plot, 2018. https://en.wikipedia.org/wiki/Boxplot/, Last accessed on 2023-04-10.

[4] Effect size, 2018. https://en.wikipedia.org/wiki/Effectsize, Last accessed on 2023-04-10.

[5] P-value, 2018. https://en.wikipedia.org/wiki/P-value, Last accessed on 2023-04-10.

[6] Mseloss — pytorch 2.0 documentation, 2023. https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html.

[7] Outlier, 2023. https://en.wikipedia.org/w/index.php?title=Outlier&oldid=1144860671.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris

Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[9] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pages 89–98. IEEE, 2006.

[10] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, 2007.

[11] Afsoon Afzal, Claire Le Goues, and Christopher Steven Timperley. Mithra: Anomaly detection as an oracle for cyberphysical systems. *IEEE Transactions on Software Engineering*, 48(11):4535–4552, 2021.

[12] Brian DO Anderson and John B Moore. *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

[13] Panos J Antsaklis. Control of cyber-physical systems. *Proceedings of the IEEE*, 2014.

[14] Shih-Ming Bai and Shyi-Ming Chen. Automatically constructing grade membership functions of fuzzy rules for students' evaluation. *Expert Systems with Applications*, 35(3):1408–1414, 2008.

[15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2014.

[16] Hamid R Berenji and Pratap Khedkar. Learning and tuning fuzzy logic controllers through reinforcements. Technical report, 1992.

[17] Dimitri Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[18] Stephen Boyd, Laurent El Ghaoui, Eric Feron, and Venkataramanan Balakrishnan. *Linear matrix inequalities in system and control theory*. SIAM, 1994.

[19] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *Journal of Systems and Software*, 164:110542, 2020.

[20] William L Brogan. *Modern control theory*. Pearson education india, 1985.

[21] Lucian Buşoniu, Tim De Bruin, Domagoj Tolić, Jens Kober, and Ivana Palunko. Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, 46:8–28, 2018.

[22] Tsong Yueh Chen, Shing Chi Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. department of computer science, hong kong university of science and technology. Technical report, Tech. Rep. HKUST-CS98-01, 1998.

[23] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys*, 51(1):1–27, 2018.

[24] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[25] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[26] Farama Foundation. Frozen lake, 2023.

[27] Farama Foundation. Mountaincar continuous, 2023.

[28] Farama Foundation. Gymnasium: A standard api for reinforcement learning and a diverse set of reference environments. `https://gymnasium.farama.org/`, 2023.

[29] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[30] Zoran Gajic and Muhammad Tahir Javed Qureshi. *Lyapunov matrix equation in system stability and control*. Courier Corporation, 2008.

[31] F Gene, J Da Powell, Abbas Emami-Naeini, Roosevelt Braun, and Jaqueline Flatley. *Feedback Control of Dynamic Systems (7th Global Edition)*. Pearson, 2015.

[32] Geoffrey J Gordon. Chattering in sarsa (lambda)-a cmu learning lab internal report. Technical report, Technical report). Carnegie Mellon University, 1996.

[33] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1418–1430, 2022.

[34] William F. Guthrie. *NIST/SEMATECH e-Handbook of Statistical Methods*. National Institute of Standards and Technology, 2020.

[35] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. A system identification based oracle for control-cps software fault localization. In *41st ACM/IEEE International Conference on Software Engineering*, pages 116–127, 2019.

[36] Zhijian He et al. Mpc system identification method based oracle for control-cps software fault localization. 2018.

[37] Thomas A Henzinger. The theory of hybrid automata. *Verification of Digital and Hybrid Systems*, 2006.

[38] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[39] Martin W Hoffmann, Somayeh Malakuti, Sten Grüner, Soeren Finster, Jörg Gebhardt, Ruomu Tan, Thorsten Schindler, and Thomas Gamer. Developing industrial cps: A multi-disciplinary challenge. *Sensors*, 21(6):1991, 2021.

[40] Junyan Hu, Barry Lennox, and Farshad Arvin. Robust formation control for networked robotic systems using negative imaginary dynamics. *Automatica*, 140:110235, 2022.

[41] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161, 2019.

[42] Eyke Hüllermeier. Fuzzy methods in machine learning and data mining: Status and prospects. *Fuzzy sets and Systems*, 156(3):387–406, 2005.

[43] Maximilian Igl, Kamil Ciosek, Yingzhen Li, Sebastian Tschiatschek, Cheng Zhang, Sam Devlin, and Katja Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. *Advances in neural information processing systems*, 32, 2019.

[44] James A Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.

[45] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[46] Kishor Jothimurugan, Rajeev Alur, and Osbert Bastani. A composable specification language for reinforcement learning tasks. *Advances in Neural Information Processing Systems*, 32, 2019.

[47] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[48] Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of zero-shot generalisation in deep reinforcement learning. *Journal of Artificial Intelligence Research*, 76:201–264, 2023.

[49] George J Klir and Bo Yuan. Fuzzy sets and fuzzy logic: theory and applications. *Possibility Theory versus Probab. Theory*, 32(2):207–208, 1996.

[50] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[51] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *2011 IEEE 36th Conference on Local Computer Networks*, pages 291–294, 2011.

[52] Takeru Kuroiwa, Yusuke Aoyama, and Noriyuki Kushiro. Testing environment for cps by cooperating model checking with execution testing. *Procedia Computer Science*, 96:1341–1350, 2016.

[53] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[54] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[55] Oleg Latypov. A comprehensive guide to proximal policy optimization (ppo) in ai, 2023.

[56] Edward A Lee. Cyber physical systems: Design challenges. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[57] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[58] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[59] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[60] Jayawant N Mandrekar. Receiver operating characteristic curve in diagnostic test assessment. *Journal of Thoracic Oncology*, 5(9):1315–1316, 2010.

[61] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[62] Thomas M Moerland, Joost Broekens, Aske Plaat, Catholijn M Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

[63] László Monostori. Cyber-physical systems in manufacturing. *CIRP Annals*, 2016.

[64] Nenad Muskinja and Boris Tovornik. Swinging up and stabilization of a real inverted pendulum. *IEEE transactions on industrial electronics*, 53(2):631–639, 2006.

[65] Roberto Natella, Domenico Cotroneo, Joao A Duraes, and Henrique S Madeira. On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, 39(1):80–96, 2012.

[66] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. *Automated software engineering*, 29(1):8, 2022.

[67] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. *Automated software engineering*, 29(1):8, 2022.

[68] OpenAI. Proximal policy optimization, 2017.

[69] Lin Padgham, Zhiyong Zhang, John Thangarajah, and Tim Miller. Model-based test oracle generation for automated unit testing of agent systems. *IEEE Transactions on Software Engineering*, 39(9):1230–1244, 2013.

[70] Qi Pang, Yuanyuan Yuan, and Shuai Wang. Mdpfuzz: testing models solving markov decision processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 378–390, 2022.

[71] Annibale Panichella and Cynthia CS Liem. What are we really testing in mutation testing for machine learning? a critical reflection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 66–70. IEEE, 2021.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8024–8035, 2019.

[73] QixinWangCpsLab. RL-testing-new: PyTorch version of Stable Baselines, reliable implementations of reinforcement learning algorithms. `https://github.com/QixinWangCpsLab/RL-testing-new`, 2024.

[74] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[75] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[76] Raghu Rajan, Jessica Lizeth Borja Diaz, Suresh Guttikonda, Fabio Ferreira, André Biedenkapp, Jan Ole von Hartz, and Frank Hutter. Mdp playground: An analysis and debug testbed for reinforcement learning. *Journal of Artificial Intelligence Research*, 77:821–890, 2023.

[77] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. *2010 47th ACM/IEEE Design Automation Conference (DAC)*, pages 731–736, 2010.

[78] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. *arXiv preprint arXiv:2005.04118*, 2020.

[79] Shankar Sastry. Lyapunov stability theory. *Nonlinear systems: analysis, stability, and control*, pages 182–234, 1999.

[80] Lui Sha, S. Gopalakrishnan, Xue Liu, and Qi Wang. Cyber-physical systems: A new frontier. *2009 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 1–9, 2008.

[81] Lui Sha, Sathish Gopalakrishnan, Xue Liu, and Qixin Wang. Cyber-physical systems: A new frontier. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 1–9, 2008.

[82] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774–788, 2011.

[83] Weijun Shen, Jun Wan, and Zhenyu Chen. Munn: Mutation analysis of neural networks. In *2018 IEEE international conference on software quality, reliability and security companion (QRS-C)*, pages 108–115. IEEE, 2018.

[84] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[85] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38:287–308, 2000.

[86] Ze Sui, Yue Zhou, Xu Zhao, Ao Chen, and Yiyang Ni. Joint intention and trajectory prediction based on transformer. In *IEEE International Workshop on Intelligent Robots and Systems*, pages 7082–7088, 2021.

[87] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[88] Martin Tappler, Filip Cano Córdoba, Bernhard K Aichernig, and Bettina Könighofer. Search-based testing of reinforcement learning. *arXiv preprint arXiv:2205.04887*, 2022.

[89] John Thickstun. The transformer model in equations. 2021.

[90] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

[91] Chris Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 2008.

[92] Mahsa Varshosaz, Mohsen Ghaffari, Einar Broch Johnsen, and Andrzej Wasowski. Formal specification and testing for reinforcement learning. *Proceedings of the ACM on Programming Languages*, 7(ICFP):125–158, 2023.

[93] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017.

[94] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[95] Xiaohui Wan, Tiancheng Li, Weibin Lin, Yi Cai, and Zheng Zheng. Coverage-guided fuzzing for deep reinforcement learning systems. *Journal of Systems and Software*, 210:111963, 2024.

[96] Mohammad Wardat, Wei Le, and Hridesh Rajan. Deeplocalize: Fault localization for deep neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 251–262. IEEE, 2021.

[97] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[98] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[99] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[100] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[101] Xiaoyuan Xie, Zhiyi Zhang, Tsong Yueh Chen, Yang Liu, Pak-Lok Poon, and Baowen Xu. Mettle: A metamorphic testing approach to assessing and validating unsupervised machine learning systems. *IEEE Transactions on Reliability*, 69(4):1293–1322, 2020.

[102] Zhangyan Xu, Shichao Shang, Wenbin Qian, and Wenhao Shu. A method for fuzzy risk analysis based on the new similarity of trapezoidal fuzzy numbers. *Expert Systems with Applications*, 37(3):1920–1927, 2010.

[103] Lotfi Asker Zadeh. Fuzzy sets. *Information and control*, 8(3):338–353, 1965.

[104] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 48(1):1–36, 2020.

[105] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. Drn: A deep reinforcement learning framework for news recommendation. In *Proceedings of the 2018 world wide web conference*, pages 167–176, 2018.

[106] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C Briand, Mojtaba Bagherzadeh, and S Ramesh. A search-based testing approach for deep reinforcement learning agents. *IEEE Transactions on Software Engineering*, 49(7):3715–3735, July 2023.