COSMOS

DevOps for Complex Cyber-physical Systems

Project Number 957254

D4.1 Prototype of a toolset supporting automated testing in co-simulated environments

Version 1.0
22 December 2021
Final

Public Distribution

University of Luxembourg

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the COSMOS Project Partners accept no liability for any error or omission in the same.

© 2021 Copyright in this document remains vested in the COSMOS Project Partners.

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Outline	15 October 2021
0.2	Background completed	30 October 2021
0.3	Approach draft	30 November 2021
0.4	Empirical results added	13 December 2021
0.5	First full draft	15 December 2021
0.8	Updates following partner reviews	20 December 2021
1.0	QA review	22 December 2021

Table of Contents

1 Introduction **1**

2 Background and related work **3**
2.1 Automated testing of Cyber-Physical Systems' Software . . . 3
2.2 Co-simulation frameworks and testing . . . 4
2.3 Metaheuristic search for CPS testing . . . 6
2.3.1 Overview of state-of-the-art genetic algorithms . . . 7
2.3.2 Genetic algorithms for CPS testing . . . 8
2.4 Reinforcement learning . . . 9
2.4.1 Reinforcement learning algorithms . . . 10
2.4.2 Reinforcement learning in software testing . . . 12

3 COSMOS CPS Testing Automation **13**
3.1 RL-based Testing Methodology . . . 13
3.1.1 Define the testing problem . . . 14
3.1.2 Train the RL model . . . 15
3.1.3 Test the SUT . . . 16
3.1.4 Analyze results . . . 16
3.1.5 Testing updated SUT versions . . . 17
3.2 TEA Architecture . . . 17
3.2.1 Connectors . . . 18
3.2.2 Properties and Actions . . . 18
3.2.3 Environment . . . 18
3.2.4 Model . . . 19
3.2.5 Testing problem . . . 19
3.3 TEA Tools . . . 21

4 Preliminary evaluation **22**
4.1 Case study 1: Autonomous flight . . . 22
4.1.1 Problem Definition . . . 22
4.1.2 Experiment . . . 25
4.2 Case study 2: Autonomous driving . . . 26
4.2.1 Problem definition . . . 26
4.2.2 Experiment . . . 27

5 Conclusion **29**

A Tutorial **30**

B API Reference **33**
B.1 Actions . . . 33
B.1.1 Interface . . . 33
B.1.2 Implementations . . . 33
B.2 Connectors . . . 35
B.2.1 Interface . . . 35
B.2.2 Implementations . . . 35
B.3 Environment . . . 37
B.3.1 OpenAI Gym Environment . . . 38
B.4 Properties . . . 40

B.4.1 Interface . . . 40
B.4.2 Implementations . . . 40
B.5 Recreating past episodes . . . 42
B.6 Statistics on past episodes . . . 43
B.7 RL model for tabular algorithms . . . 47
B.8 Testing problem . . . 49
B.9 Utility classes and functions . . . 50

C FlightGear properties **52**
C.1 Selected properties . . . 52
C.2 All tested properties . . . 56

Executive Summary

This deliverable concerns the development of the COSMOS solution for the automated system and component-level testing of software running in cyber-physical systems (CPS). The developed solution aims to automate functional testing (i.e., verifying that the software meets its functional requirements) and testing for functional safety.

We present a black-box automated testing solution that does not require models of the system nor implementation details. To test the CPS software under representative conditions we rely on simulators. Since CPS software often needs to react to external agents (animated and inanimate ones), we focused on the development of a solution that relies on reinforcement learning, which is an AI-based algorithm that learns to interact with other agents like an animated actor would do. In our context, reinforcement learning is used to simulate an environment that is adversary to the CPS software under test (i.e., an environment that aims to make the software fail).

We have performed a preliminary evaluation with two open source case studies that rely on the same simulators used by some of the COSMOS use case provider. Preliminary empirical results have shown that the proposed approach can effectively make the software under test fail; also, it showed venues for the improvement of the tools developed to support debugging of the failures observed. We also provide a tutorial and our API reference.

1 Introduction

This deliverable concerns COSMOS work about the development of scalable and effective solutions for performing V&V for cyber-physical systems (CPS) when executed in DevOps pipelines. In particular, it concerns the development of automated software testing techniques, which are techniques that can (1) automatically derive the inputs to cost-effectively exercise the software under test and (2) automatically determine the test outcome (i.e., pass or fail); the latter based on high-level specifications provided by the end-user. Our automated software testing techniques leverage machine learning and simulated environments to effectively exercise CPS in DevOps contexts, which is one of the objectives of WP4. The developed techniques aim to automate functional testing (i.e., verifying that the software meets its functional requirements); in the CPS context, functional testing relates to testing functional safety, which is the capability of a safety-critical system to correctly respond to inputs and behave in a safely manner. This deliverable aims to detail the progress made in the first 12 months of the COSMOS project.

With the term *simulated environment* or, simply, *simulator* we refer to *a device, computer program, or system that performs simulation*, with simulation being *the process of using a model that behaves or operates like a given system when provided a set of controlled inputs [1].* According to the COSMOS project proposal, this deliverable was initially planned to concern solutions for testing automation within *co-simulation environments*[1] by relying on a model-based approach. However, the requirement analysis phase of COSMOS has shown that representative companies in the CPS domain (i.e., COSMOS use case providers) often do not rely on model-driven development practices (for example, they may rely on models in early design stages but such design models are not kept up-to-date during the development of the system). Also, many companies do not need general-purpose co-simulation environments; indeed, they often rely on domain-specific simulators that integrate all the features required for testing their products. Simulation is thus a necessity when testing CPS software but the type of simulators being used may vary from company to company. For example, COSMOS use case providers rely on simulators that range from PikeOS [49], which includes a CPU emulator[2] and enables simulation of I/O channels, to FlightGear [19], which simulates the dynamics of airplanes and the environment, to CARLA [14] and BeamNG [10], which can simulate vehicles on the road.

Companies that develop CPS software need automated software testing solutions that can be used across CPS software that largely differ in architecture and programming constructs adopted. Such diversity is well represented by COSMOS use case providers whose products range from embedded software installed on dedicated boards (Unparallel Innovation), to networking applications for railway systems (Qmedia), to traffic control systems (GMV), to automotive frameworks (AICAS). To cope with a wide range of design choices and programming languages, we focused on the development of a *black-box automated testing solution*, which is often the case in CPS literature. Also, since CPS software often needs to react to external agents (animated and inanimate ones), we focused on the development of a solution that relies on *reinforcement learning*. Our motivation lies on reinforcement learning aiming at learning how to interact with an environment towards a goal (in our context, make the software fail). Such characteristics shall make reinforcement learning more effective than algorithms based on metaheuristic search (i.e., the state-of-the-art in software testing research). Finally, our solution does not make assumptions on the type of simulator being used, it simply requires it to be controllable through an external interface. Being black-box and based on generic simulators, our solution shall enable testing of both whole software systems and software components. Also, it shall be usable with target hardware in the loop.

We have performed a preliminary evaluation with two opensource case study subjects focused on flight autopilot and autonomous driving, respectively. We selected case study subjects that rely on the same simulators used by some of the COSMOS use case provider (i.e., GMV and AICAS); our choice shall facilitate integra-

[1] Co-simulation is *the theory and techniques to enable global simulation of a coupled system via the composition of simulators* [25].

[2] The term *emulation* refers to the process of using a model that accepts the same inputs and produces the same outputs as the system represented. For example, one can emulate an 8-bit computer with a 32-bit computer [1].

tion in later stages of the project. Preliminary empirical results have shown that the proposed approach can effectively detect failures; also, it showed venues for the improvement of the tools developed for results analysis (i.e., to support debugging of the failures observed). Moreover, solutions to make testing more efficient (e.g., to fit into short DevOps cycles) will be developed in future months by relying on the integration of additional machine learning solutions. Future work will also concern the extension of the empirical evaluation, including the application of the developed approach to COSMOS use cases that rely on simulators for functional testing, which are the ones provided by GMV (Avionics), AICAS (Automotive), QMedia (Railway), and Unparallel Innovation (Utility management); such evaluation will also concern studying the feasibility for the developed approach to deal with testing for non-functional requirements.

This deliverable proceeds as follows. Section 2 presents background and related work. Section 3 describes the COSMOS testing automation approach. Section 4 provides an overview of our preliminary case study. Section 5 concludes the deliverable.

Appendix A provides a tutorial for using our toolset. Appendix B provides our API reference.

We plan to release the toolset under an open-source license according to the exploitation plans of the project. For the time being, the software prototype is available to COSMOS reviewers at the following URL `https://gitlab.uni.lu/cosmos/simulator-based-testing`[3]

[3]Please contact the consortium representatives and/or `fabrizio.pastore@uni.lu` to request access (GitHub account handle required).

2 Background and related work

In this Section, we position our contributions with respect to state-of-the-art solutions for testing CPS software and we motivate our choice for proposing a black-box approach (Section 2.1); we provide an overview of co-simulation frameworks, describe the lack of automated testing solutions for that context, and clarify why they are not used in COSMOS (Section 2.2); we describe metaheuristic search algorithms, a common black-box testing solution for CPS software (Section 2.3); finally, we provide an overview of reinforcement learning and its use in software testing research (Section 2.4).

2.1 Automated testing of Cyber-Physical Systems' Software

Cyber-Physical Systems (CPSs) are heterogeneous systems that integrate computation, networking, and physical processes that are deeply interlaced [29]. In CPS software, conformance with requirements is often verified through testing executed at different development stages, based on available development artifacts [42].

Depending on the development stage, testing of CPS software is performed through simulators or with hardware in the loop. CPS software testing consists of providing specific 'inputs' to a system, executing the system with those inputs, and examining system 'outputs' or 'behaviours' for conformance to specifications. Pure simulation occurs in the model-in-the-loop (MiL) and software-in-the-loop (SiL) stages where the environment, the physical system (i.e., the plant), and the software components (in the case of MiL) are simulated. Hardware is used both (1) when the system is validated on the final environment and (2) in the hardware-in-the-loop (HiL) simulation where the software components of the CPS are deployed on the final hardware while the plant and the environment are simulated. The choice of the simulation environment and the simulation strategy varies across development teams. When the process is not model-driven, testing starts in the SiL stage and may rely on domain-specific simulators (e.g., FlightGear for UAVs); SiL, MiL, and HiL simulations are performed using a single simulation environment (e.g., Matlab/Simulink) when such environment supports all the required modelling languages. In contrast, co-simulation frameworks are used when diverse modelling languages shall be used (e.g., SysML and Simulink, see Section 2.2).

The interest in the development of solutions that automate testing of CPS software is recent; indeed, only a few recent surveys on the topic exist [45, 8, 56, 51]. Zhou et al. [56] classify test automation approaches according to their underlying paradigm (i.e., model-based, search-based, fault-injection-based and less related approaches such as Cloud-based and runtime verification), their objectives (i.e., conformance testing, robustness testing, fragility testing, and security testing). In their survey, Zhou et al. provide an overview of the environments used to test CPS software (i.e., the testbeds) and highlight that most CPS testbeds rely on simulation, with Simulink [35], OPAL-RT [37], RTDS [43] being the most common technologies. Simulink is mostly used for MiL simulations while OPAL-RT is a PC/FPGA-based real-time simulator targeting HiL testing. The importance of simulation in CPS testing is also highlighted by the high number of simulator-based approaches appearing in the survey of Asadollah et al. [8]. Another characteristic of testing automation techniques (both model-based and search-based) is that they are black-box, which means that they do not rely on the analysis of the software (source code or executable). This is essential for complex systems including third party components whose source codes are not accessible.

The findings reported above are confirmed by the survey of Sadri-Moshkenani et al. [45], who focused on techniques for test case generation, selection and prioritization for CPSs — topics that are related to continuous integration and continuous development practices. Sadri-Moshkenani et al. surveyed 34 techniques, all of them being model-based, with 15 of them deriving test inputs statically and 19 dynamically. Concerning the *simulation context*, 32 techniques are applicable to MIL, 30 to SIL, and 22 to HIL; in general, the authors stress that the black-box nature of the surveyed techniques enables their applicability to different contexts (i.e., SIL, MIL, or HIL), which is a characteristic that we also aim to leverage in COSMOS. Concerning the *type* of technique, 26 techniques concern the generation of test inputs, six techniques concern test case selection,

seven test case prioritization; some techniques have more than one purpose (e.g., selection and prioritization). Authors stress that no techniques automate regression testing (i.e., determine if the performance of a feature available in a previous version worsened), which is, instead, one of the contributions of COSMOS. Concerning their *objectives*, 26 techniques aim to improve testing efficiency, seven techniques aim to improve testing effectiveness (i.e., fault revealing capability), and only six concern safety. Such numbers highlight the need for approaches that target test effectiveness and safety, which are our objectives in COSMOS. Finally, only five of the surveyed papers provide links to publicly available tools, which highlights the need for automated solutions such as those proposed by COSMOS.

Sadri-Moshkenani et al. [45] point out that no code-based generation, selection, or prioritization approach has been proposed despite the fact that some degree of manual coding is still required to fully implement CPS software, even though some models (e.g., Simulink models) can be used to automatically generate CPS code. However, we believe that white-box approaches are likely discarded because of the many limitations of techniques based on combining static analysis and constraint solving (e.g., limited support for floating-point arithmetic in symbolic execution tools [23]). Furthermore, alternative approaches, referred to as fuzzers, which rely on dynamic program analysis (typically, the analysis of code coverage) to drive the generation of test inputs, have been applied only to simplified CPS prototypes or CPS units with few lines of code [15, 46] — likely because of the difficulty of monitoring large systems with integrated components. For these reasons, in COSMOS, we will largely focus on the black-box generation of test inputs using simulators, a characteristic shared with the approaches described in the survey above. However, different from most of the work mentioned by Sadri-Moshkenani et al., we will not rely on models of the system under test to select test inputs; this choice will enable COSMOS to target CPS software that is not developed according to model-driven engineering (MDE) principles. Supporting systems not developed through MDE practices is important for three reasons. First, not all the CPS developers rely on MDE (e.g. because they are not using modelling languages that can be automatically translated into executable code). Second, some CPS components, like components integrating deep neural networks, can only be tested in a black-box fashion. Third, as highlighted also by Sadri-Moshkenani et al., some degree of manual coding is usually required even when adopting MDE practices. Finally, our solution may still enable testing software developed according to MDE principles; it might be useful, for example, to integrate a model-based (e.g., coverage driven) test suite with our testing strategy, which observe the behaviour of the software and act accordingly (see Section 3).

2.2 Co-simulation frameworks and testing

Gomes et al. defined a comprehensive taxonomy of existing co-simulation techniques [25]. They define co-simulation as *the theory and techniques to enable global simulation of a coupled system via the composition of simulators*. They classify existing techniques into three main types: (1) discrete-event-based co-simulations (e.g., a traffic light), (2) continuous-time-based co-simulations (e.g., mass-spring-damper in a car suspension system), and (3) hybrid co-simulations (mixing the characteristics and assumptions of discrete and continuous systems, e.g., a thermostat regulating the temperature in a room). Discrete-event-based systems are characterized by *reactivity* (instant reaction to external stimuli) and *transiency* (the system can change its state multiple times in the same simulated time point, and receive simultaneous stimuli). A continuous-time system is assumed to have a state that evolves continuously over time. The authors describe co-simulation challenges, including semantic adaptation, modular coupling, stability and accuracy, and finding a standard for hybrid co-simulation. However, the survey does not cover the topic of testing automation through co-simulation.

The most recent evaluation of co-simulation frameworks from a testing perspective is that of Gonzalez et al. [26], who evaluated five frameworks, including TASTE [40], INTO-CPS [33], OpenMETA [50], Cosimate [16] and CyPhySim [13]. For their evaluation, they considered five criteria: efficiency, support for co-simulation with SysML-Simulink, degree of automation (i.e., limited need for user intervention), controllability, and observability. Table 1 provides a summary of the results reported by Gonzalez et al., with some of the entries being updated according to the recent developments of the considered tools. Gonzalez et al.

Table 1: Co-simulation frameworks analyzed in related work

	MODELS 2018 paper	INTO-CPS*	TASTE	OpenMETA	CyPhySim	CosiMate*
URL	Available to COSMOS	into-cps.org	https://essr.esa.int/project/taste	None	ptolemy.berkeley.edu/ projects/chess/cyphysim/	cosimate.com
Year	2018	V4.0.1 Jan 2021	2021	Paper in 2014	Paper in 2015	V2020.01 Jan 2020
Type	Co-simulation framework	Integrated toolchain for model-based design of CPS.	Open-source toolchain for embedded, real-time systems	Tool suite for component and model based design of engineered systems.	CPS simulator	Co-simulation platform
Efficiency	High	Depends on what is embedded in the FMUs (complied code, model solvers, etc.).	High	Medium	Depends on what is embedded in the FMUs (complied code, model solvers, etc.)	N/A
Synchronized execution	High (automatically generated helper class which interacts with the MatLab runtime)	Medium (support SysML but not Simulink)	Medium	Low	Medium	N/A
Degree of automation	High (automatic execution)	Low (user must perform some manual work)	Low (user must perform some manual work)	Low (user must perform some manual work)	Low (user must perform some manual work)	N/A
Controllability	High (based on static cyclic scheduler)	High (it is possible to simulate scheduling algorithms)	High (it is possible to simulate scheduling algorithms)	Low	Medium	N/A
Observability (execution traces)	High (traces with high level of detail)	Low (does not include sequences of operation calls and data flows).	Low (traces without timestamps)	Low (data comes from an external simulator)	Low (manual work to model the outputs from the functional models – similar to Simulink)	Integrated debugging and monitoring features (Spy data, VCD and Excel traces, co-simulation controls).
Modelling Languages	SysML for software and hardware (structural and behavioral modeling of CPS) – supported by a profile to build testable models of CPS. Simulink for hardware and environment (function models).	SysML, OpenModelica, Vienna Development Method (VDM) formal language	AADL (Architecture Analysis & Design Language), ASN.1 (Abstract Syntax Notation One), and SDL (Specification and Description Language)	Heterogeneous models (in Simulink and OpenModelica), composed to offer a unified representation of the system.	Ptolemy II, a modeling and simulation framework for heterogeneous models. Combines state machines for behavioral modeling, and the actor-oriented language of Ptolemy II for structural modeling	Simulink, HDL, UML, FMI, Modelica C, C++, Java, VHDL, VHDL-AMS
Models Exported to FMI/FMUs	No	Yes (FMUs)	No	Yes (FMI)	Yes	FMI
Co-simulation Engine	Co-simulation framework based on SysML and Simulink. It uses MatLab runtime to execute the models.	Co-simulation Orchestration Engine (COE) that supports the co-simulation of structural and behavioral software models described in VDM.	None	When the unified model is executed, OpenMETA seamlessly invokes external simulation tools (Simulink and OpenModelica) to execute the composed models.	Compliant simulation environments can import and instantiate the FMUs to enable co-simulation.	Supports heterogeneous co-simulation between solvers (e.g., Simulink) and event-driven (HDL, UML) or sequential (C) simulators.
Code Generation	SysML models are passed to a code generator that produces C++ code. External libraries are manually integrated.	The code generation and deployment capabilities of 20-sim are used for HiL testing.	It uses the information in the models to generate skeleton code, that must then be completed manually.	No	No	N/A
Output	Traces	Traces when the co-simulation is run.	When the code is run, execution traces can be recorded.	A unified model.	FMUs	N/A
Testing Support	Yes, randomly generated test cases (path coverage on SysML ADs)	Yes, RTT-MBT	Automatic generation and execution of Python test scripts.	Mechanisms to define testing scenarios, whose execution can be automated.	No	Yes

Legend: N/A not discussed in related-work. (*) updated by the authors of this deliverable based on information available at the time of the writing.

highlighted that, at the time of their analysis, co-simulation techniques did not support the execution of architectural and behavioral models or lacked observability and controllability of tasks scheduling. They proposed a SysML-based modeling methodology for the construction of testable models of CPSs and a SysML-Simulink co-simulation framework (with controllability and observability features) to allow model testing of CPSs.

Among the tools considered by Gonzalez et al., INTO-CPS is the toolset with the largest support from an active academic community; in 2020, the University of Luxembourg (UL) joined the INTO-CPS association to access the toolset and evaluate it in comparison to the tool from Gonzalez et al., which is a preliminary prototype developed by UL. In general, although the current release of INTO-CPS supports a larger set of modelling languages than the prototype of Gonzalez et al., they both do not provide a mature solution to support automated, black-box testing. More precisely, the prototype of Gonzalez et al. does not integrate an automated testing solution, while the INTO-CPS toolset includes a prototype testing tool, which has been released with a commercial licence[4]. The INTO-CPS testing tool is called *RT-Tester Model Based Test Case and Test Data Generator (RTT-MBT)*, it relies on the *RT-Tester* framework to automate test cases execution while it relies on a model-based approach to derive test inputs; the available documentation for RTT-MBT

[4]https://www.verified.de/products/model-based-testing/

indicates that it aims to achieve structural coverage of the model [6], a detailed description of updated features is not available to the general public[5].

In general, co-simulation frameworks target software systems developed according to MDE practices, which are not the main target for COSMOS. Indeed, COSMOS aims to target both CPS software systems developed according to MDE practices and software developed not following a model-driven approach. Also, based on the requirements defined by COSMOS use case providers, the problems addressed by co-simulation frameworks are not a priority for the project consortium. For the reasons above, the development of a testing solution targeting co-simulation frameworks had been descoped from the project. Instead, COSMOS will develop a generic black-box testing solution, which aims at supporting any simulator that provides an API for controlling its execution. The COSMOS solution might be also integrated with co-simulation frameworks if these provide an interface to control the different integrated simulators.

In addition to generic frameworks enabling co-simulation of CPS, the literature reports also on ad-hoc co-simulation solutions for domain-specific simulators; in particular, we report about work for FlightGear [9], the simulation framework used by GMV to test their software, which has been considered in the research literature (work about simulators for the automotive context are discussed in Section 2.3). Aschauer et al. presented a simple co-simulation solution for Matlab and FlightGear [9]. They used UDP for communication between the two programs. Flight measurements (flight attitude data) from FlightGear are sent to Matlab, which uses a controller model to compute input values for the aircraft actuators (throttle, elevator, aileron, and rudder) and then the computed values are sent back to FlightGear. They used three state-space controllers (a model predictive controller, a linear quadratic regulator, and a gain-scheduled linear quadratic controller) to control the altitude and heading of the aircraft[6], and compared the robustness of the three controllers under turbulent weather conditions. Another solution is GaTAC, which is a simulation framework for evaluating autonomous control in UAVs [47]. It can be used to simulate and evaluate policies generated by multi-agent decision-making algorithms and to facilitate the development and evaluation of scalable decision-making algorithms. GaTAC is open-source and is implemented in C++. It uses FlightGear as a flight simulator. It deploys multiple instances of FlightGear on a computer network using a scalable architecture. GaTAC establishes UDP sockets to communicate with FlightGear instances. It controls UAVs by sending control data at a low level (more than 30 parameters including throttle, rudder, elevator, and aileron) to FlightGear. Also, it collects from FlightGear data such as current latitude, longitude, altitude, values of different surfaces, and fuel level. Different from the solutions described above, COSMOS will not provide technology that works with one specific simulator but with any simulator that provides an API to control the simulation.

2.3 Metaheuristic search for CPS testing

Metaheuristic search algorithms are state-of-the-art approaches to find solutions for multi-objective optimization problems [44, 2]; they include, among others, Evolutionary Algorithms, Swarm intelligence, Artificial Immune Systems, Metaphor Based Metaheuristics (e.g., Simulated Annealing), Non-Metaphor Based Metaheuristics (e.g., Tabu Search), and [2]. In software engineering research, evolutionary algorithms (more specifically, genetic algorithms) are widely adopted to identify test inputs that maximize some coverage criterion [39]. Also, they have been successfully used to test safety-critical systems, in particular autonomous driving software [27, 11]. In the following, after a brief overview of the most popular metaheuristic search algorithms in software engineering research, we discuss how the different algorithms have been adopted to support black-box testing of CPS' software or safety-critical software components.

[5]Updated features are listed at the following URL: `https://www.verified.de/category/releases/rtt-mbt/`; however, there is no mention of their integration with INTO-CPS framework, which was lastly documented for RT-Tester V6.0-6.3.0 in 2016.

[6]Please note that the controller does not control either the altitude or the heading directly; instead, it controls parts of the aircraft, such as the rudder, the ailerons, and the elevator, which have an impact on the aircraft's attitude (i.e. its orientation in space) and thereby on its heading.

Require: O = o1, o2, .. the of number of objectives
Require: M, population size
Require: T, search budget (i.e., number of iterations to perform)
Ensure: a population whose Pareto front indicate the optimal solutions
1: $H \leftarrow$ *IDENTIFY-REFERENCE-POINTS*
2: $t \leftarrow 0$ //current generation
3: $P_t \leftarrow$ *RANDOM-POPULATION*(M)
4: **while do**$t < T$
5: $Q_t \leftarrow$ *GENERATE-OFFSPRING*(P_t)
6: $R_t \leftarrow P_t \cup Q_t$
7: $(F_1, F_2, ..., F_l) \leftarrow$ *RANKED-SORT*(R_t)
8: $r \leftarrow 0$ //selected Pareto front rank
9: **while** $|P_{t+1}| + |F_r| \leq M$ **do**
10: $P_{t+1} \leftarrow P_{t+1} \cup F_r$
11: $r \leftarrow r + 1$
12: **end while**
13: **while do**$|P_{t+1}| < M$
14: $ind \leftarrow$ *SELECT-BASED-ON-DISTANCE*(F_r, H)
15: $P_{t+1} \leftarrow P_{t+1} \cup ind$
16: **end while**
17: $t \leftarrow t + 1$
18: **end while**
19: **Return** P_t

Figure 1: NSGA-II / NSGA-III

2.3.1 Overview of state-of-the-art genetic algorithms

NSGA-II [18] is a state-of-the-art solution for optimization problems with up to three objectives (often referred to as *multi-objective* optimization problems); it is described by the algorithm in Figure 1. NSGA-II works by first generating a random population (Line 3), which is evolved in a number of iterations (Lines 4 to 18). A new offspring (Q_t) is generated by applying mutation and crossover operators (Line 5). The algorithm preserves elitism since the new population (P_{t+1}) is generated by selecting the best individuals in the union of the current population (P_t) and the offspring (Q_t) (Line 6). Selecting the best individuals is performed by ranking individuals based on their belonging to the nondominated fronts (Line 7), which is efficiently performed by the fast-nondominated-sorting procedure [18]. Then, the algorithm adds individuals to the new population (P_{t+1}) according to their rank (Line 10), till it finds a ranked subset that cannot be fully added (Line 9). Finally, to preserve diversity in the population, NSGA-II selects the items belonging to the last ranked subset according to a crowding distance function that prioritizes individuals being further away from others (Line 14).

NSGA-III is a state-of-the-art many-objectives optimization algorithm that extends NSGA-II. The term *many-objective* is used for algorithms that tackle problems with more than three objectives. To solve problems with a Pareto-optimal front whose objective values may be differently scaled, it normalizes objectives' scores. To implement an efficient alternative to crowding distance, which is expensive for a large number of objectives, it relies on the identification of H reference points that should capture an ideal Pareto front. The identified reference points are uniformly distributed on the hyper-plane and equally inclined to all objective axes [17]. At every iteration, as for NSGA-II, individuals are added to the next population by relying on nondominated sorting. For the last front, instead of relying on crowding distance, NSGA-III selects the individuals that are close to a reference point (Line 14 in Figure 1). More precisely, NSGA-III determines the reference point closer to an individual based on the distance of the individual from the vector that joins the reference point to the origin. Then it prioritizes the reference points with fewer individuals in their neighbourhood (this is done by associating each individual to one reference point, based on distance, and counting the number of individuals for each reference point). So far, NSGA-III has not been widely adopted by the software engineering community.

A popular many-objective algorithm used in software testing research is MOSA [39]. In MOSA, individuals are test cases (i.e., sequences of inputs). Since MOSA models branch coverage as a search objective but not the length of individuals, it extends NSGA-II by relying on an archive that is used to keep track of the shortest test cases accidentally identified during the search (shortest test cases are easier to read than longer ones). In addition, since MOSA aims to fulfil all the objectives and not only a subset of them, instead of relying only on

nondominace relation to select elitist individuals (i.e., Line 7 in Figure 1), it relies on the *preference criterion*, which selects for rank 0 also the individuals that minimize the objective score for the uncovered objectives (i.e., a score of zero has not been observed).

2.3.2 Genetic algorithms for CPS testing

The work of Ben Abdessalem et al. focused mainly on the problem of efficiently finding safety issues when testing Advanced Driver Assistance Systems (ADAS) in a simulated environment [3, 4]. They presented a multi-objective search approach that combines NSGA-II with neural networks as surrogate models to guide testing towards the most critical behaviors of ADAS [3]. Surrogate modeling enables the testing approach to explore a larger part of the input search space within limited computational resources. The same authors also presented a simulation-based testing algorithm for vision-based control systems that builds on learnable evolutionary algorithms [4]. In this case, NSGA-II is combined with decision tree classification models to find critical test scenarios and critical regions in the input space for ADAS. Finally, they also addressed the challenge of detecting and managing feature interactions that violate system requirements, leading to failures [12]. They propose FITEST, which extends and adapts MOSA to testing feature-based self-driving systems.

Gambi et al. proposed to combine procedural content generation based on polylines, a technique commonly employed in modern video games, and a search-based algorithm to automatically create challenging virtual scenarios for testing self-driving car software [22, 21]. Their prototype tool, called AsFault, generates virtual roads for testing lane keeping features in autonomous cars (e.g., to generate scenarios that cause the car to drive off the road). Other work, instead, takes inspiration from real-world scenarios to generate scenarios (e.g., roads in this case) to be considered for testing [36, 31].

Riccio and Tonella [41] introduced the notion of frontier of behaviors, that is, the inputs at which a Deep Learning (DL) system starts to misbehave. They proposed a search-based tool, called DeepJanus, that combines a multi-objective evolutionary algorithm (NSGA-II) with novelty search to generate frontier inputs for DL systems. DeepJanus characterizes the frontier of DNN misbehaviours by identifying pairs of inputs that are close to each other, with one input leading to a correct DNN result and the other to a DNN error. It relies on the popular NSGAII algorithm extended with an archive (to keep the best individuals found in the search) and with repopulation (to escape from stagnation by replacing the most dominated individuals with random ones). Deepjanus relies on a fitness function that includes a measure of the sparseness of the solutions. Sparseness is measured as the distance from the closest input in the archive. They considered a self-driving car that was trained and tested in the BeamNG simulation environment. The experimental results obtained for a Lane Keeping Assist System component of a self-driving car showed that the frontier of a well-trained system contains almost exclusively unrealistic roads that violate the best practices of civil engineering, while the frontier of a poorly trained one includes many valid inputs that point to serious deficiencies of the system.

Ul Haq et al. studied the relationship between offline and online testing for Deep Neural Networks (DNNs) [28]. In offline testing, DNNs are tested as individual units based on test datasets obtained independently from the DNNs under test, while in online testing DNNs are embedded into a specific application environment and are tested in a closed-loop mode in interaction with the application environment. To perform the study, Ul Haq et al. rely on random generation of scenarios. As the system under test, they used an automated driving system, in which DNNs were used to automate end-to-end controls of steering functions of a self-driving vehicle. Their results showed that offline testing is more optimistic than online testing as many safety violations identified by online testing could not be identified by offline testing.

The papers presented above demonstrated the feasibility of relying on metaheuristic search algorithms to efficiently test a CPS component in a black-box manner. However, genetic algorithms (i.e., the widely adopted solution in software testing) are limited by the need to represent the whole sequence of inputs for the simulator software (usually referred to as *test scenario*) as a solution to the optimization problem. Also, metaheuristic search algorithms, more in general, may be highly inefficient when generating input sequences of arbitrary length. We clarify these limitations in the following.

In the case of genetic algorithms, there is no flexibility on the number of inputs to be provided to the software under test. For example, Ben Abdessalem et al. define the *test scenario* with a tuple (S, O, I, D, C), where $S = s_1, ..., s_n$ is a set of variables specifying (immobile) static environment aspects (e.g., road shape). O is a set of mobile objects (pedestrians and vehicles). $I = i_1, ..., i_m$ is a set of variables specifying initial states of the mobile objects in O (e.g., the speed of a pedestrian). Each variable in I is related to one mobile object in O, while each mobile object $o \in O$ is related to one or more variables in I. D is a set of domains of values for variables in $S \cup I$. C is a set of boolean propositional constraints over $S \cup I$. The test scenario is first generated by the evolutionary algorithm and then tested using the simulator. Such characteristic requires determining the number of inputs belonging to a test scenario when configuring the algorithm. For example, in the work of Ben Abdessalem et al. vehicles cannot change their speed within a scenario, as additional variables would otherwise be required (e.g., a vector capturing speed at instant t).

Constraining the number of inputs belonging to a test scenario may reduce testing effectiveness, especially when the software under test interacts with agents that may change their state often, which is typical for CPS. For example, this is the case of UAVs flying in the presence of other UAVs or simply in the presence of weather conditions that may change abruptly. Metaheuristic search solutions relying on ad-hoc algorithms [22, 21] may enable the generation of sequences with a varying number of inputs; however, the generation of long sequences of inputs through evolutionary algorithms may still be highly inefficient. For example, a large proportion of the test inputs belonging to a test scenario may be shared among the different scenarios generated by the search algorithm prior to finding an optimal one (e.g., the one that detects a fault); that case, a lot of time would be wasted performing almost the same sequences of actions. For this reason, in COSMOS, we focus on investigating algorithms that can select test inputs online, while the software is executed. According to Ul Haq et al., online testing (i.e., validating the behaviour of software within a sequence of interactions with its application environment instead of a single action) is more effective than offline testing in finding faults; we thus believe that choosing the actions to perform online (i.e., during testing) should improve test effectiveness further. To achieve our objective, we focus on reinforcement learning algorithms, which are presented in Section 2.4.

2.4 Reinforcement learning

Black-box approaches for testing CPS software aim at identifying unsafe test scenarios, that is, test scenarios in which the CPS software fails to fulfil its (safety) requirements.

Since a CPS interact with agents that may change their state during a test scenario (e.g., weather conditions, wind direction, or vehicle's speed in a scene), we believe that, in order to maximize test effectiveness (i.e., the number of faults being detected), software testing shall focus on how well the CPS software under test (hereafter referred to as the *SUT*) handles such state changes. To find faults, we shall make such agents interact with the SUT as adversaries; precisely, they shall react (i.e., change their state) based on the performance of the SUT, aiming at making it fail. For this reason, reinforcement learning appears to be an appropriate solution to test CPS software. In the following, we provide an overview of reinforcement learning algorithms and describe their adoption in software testing research.

2.4.1 Reinforcement learning algorithms

Reinforcement learning solutions aim at *learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. Reinforcement learning is a third machine learning paradigm, alongside supervised learning and unsupervised learning* [48]. Reinforcement learning relies on goal-seeking agents that can sense aspects of their environments and can choose actions to influence their environments.

Reinforcement learning consists of four main components:

- Policy: A policy defines the way the agent behaves at a given time step. It is a mapping from perceived states of the environment to actions to be taken in those states.
- Reward: A reward defines the goal of a reinforcement learning problem. It determines the immediate, intrinsic desirability of environmental states.
- Value function: It specifies what is good in the long run. The value of a state is the total sum of rewards that the agent can expect to get in the future if it starts from that state.
- A model of the environment (optional): It mimics the behavior of the environment. Models are used for planning – a way of deciding on a course of action by considering possible future situations before they are experienced. Model-free methods are simpler trial-and-error methods. Model-based methods are more sophisticated as they may involve learning a model of the environment (e.g., using a Deep Neural Network) and then using that model for planning.

Figure 2: Reinforcement learning applied to testing of CPS software

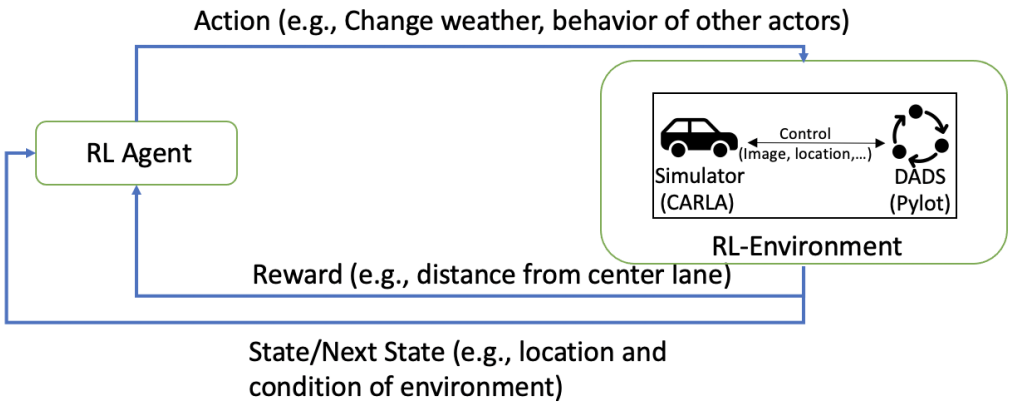


Figure 2 shows how reinforcement learning can be applied in the context of black-box testing based on simulators. The reinforcement learning agent (RL Agent) at every time step (configured by the engineer) performs an action (e.g., set a parameter of the simulator to simulate rain). The state of the reinforcement learning environment is captured after the execution of the action and is provided to the RL Agent. Based on state information, the reward is measured (e.g., the distance of a self-driving car from the center lane). A sequence of actions is called an *episode* — it might have a predefined duration (e.g., a specific number of seconds) or it may depend on the testing objectives (e.g., reaching a desired destination for a self-driving car).

Reinforcement learning methods can be classified into two broad categories: *Tabular solution methods* and *Approximate solution methods*.

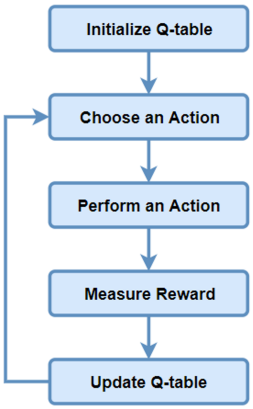
The main difference between tabular methods and approximate methods is that tabular methods require the state of the system to be discretized, which implies that the effectiveness of the approach depends on the discretization strategy. Approximate methods, instead, leverage supervised learning methods (e.g., Artificial Neural Networks, Kernel methods, Deep Neural Networks); for this reason, approximate methods require a larger execution budget to converge. In practice, tabular methods learn faster than approximate methods how to

interact with the environment (e.g., how to change environment settings to make the SUT fail, in our context). For these reasons, under the assumption that it is possible to appropriately discretize the system state, tabular methods are a reasonable choice in a software testing context, which is our focus in this document. Because of their relation with DNN and machine learning solutions, we leave the investigation of approximation methods to deliverable D4.7 (i.e., *Prototype of a toolset for machine learning- enabled detection of vulnerabilities in CPS*).

Tabular methods include (1) *dynamic programming methods*, which require a complete and accurate model of the environment, (2) *Monte Carlo methods*, which are conceptually simple and do not require a model but are not well suited for step-by-step incremental computation, and (3) *temporal difference methods*, which do not require a model and are fully incremental but are more complex to analyze. In our context, temporal-difference methods appear to be the best choice. Indeed, they do not require a model of the environment and they bootstrap (i.e., update estimates without waiting for the termination of an episode). Temporal-difference methods wait until the next time step to update the value estimates. In contrast, Monte Carlo methods must wait until the end of an episode to update the value estimates.

Two popular tabular methods are Q-learning and SARSA. They both learn a policy $Q : S \times A \rightarrow R$, which takes a (state, action) pair and returns a value (i.e., the expected sum of future rewards). The Q policy is implemented through a table that is referred to as Q-table.

Figure 3: Q-learning process



The Q-learning algorithm process is depicted in Figure 3. Before learning begins, the Q-table is initialized according to some strategy chosen by the engineer. For example, high initial values (i.e., optimistic initial conditions) encourage exploration [48]; alternatively, the first reward can be used to reset the initial conditions.

At each step i the RL agent selects an action A_i, it observes a reward R_{i+1}, and enters a new state S_{i+1}. Q is then updated. The core of the algorithm is the following Bellman equation:

$$Q(S_i, A_i) \leftarrow Q(S_i, A_i) + \underbrace{\alpha}_{\text{learning rate}} \left(R_{i+1} + \underbrace{\gamma}_{\text{discount factor}} \cdot \max_a Q(S_{i+1}, a) - Q(S_i, A_i) \right) \tag{1}$$

where R_i is the reward received when moving from the state S_i to the state S_{i+1}, and α is the learning rate ($0 < \alpha \leq 1$). The learning rate determines to what extent newly acquired information overrides old information. The discount factor γ determines the importance of future rewards. The formula is used to perform a value iteration update, using the weighted average of the old value and the new state information. Note that this equation does not depend on the action that will actually be taken at the next step, A_{i+1}, but rather on the action estimated to bring the highest future rewards, $\max_a Q(S_{i+1}, a)$. Thus Q-Learning is an *off-policy* algorithm (i.e., the policy for choosing the next action is different than the learned policy).

SARSA's value iteration update is the same as Q-learning, except for using $Q(S_{i+1}, A_{i+1})$ instead of $\max_a Q(S_{i+1}, a)$, which leads to the following:

$$Q(S_i, A_i) \leftarrow Q(S_i, A_i) + \underbrace{\alpha}_{\text{learning rate}} \left(R_{i+1} + \underbrace{\gamma}_{\text{discount factor}} \cdot Q(S_{i+1}, A_{i+1}) - Q(S_i, A_i) \right) \quad (2)$$

SARSA is on-policy (i.e., the policy for choosing the next action is equal to the learned policy). Indeed, a SARSA agent interacts with the environment and updates the policy based on the actions taken (i.e., $Q(S_{i+1}, A_{i+1})$). Q-learning updates the Q value according to the best estimate (i.e., $\max_a Q(S_{i+1}, a)$). The difference between the two algorithms derives from the fact that the action taken at step i is selected according to an ϵ-greedy policy, which means that most of the time the algorithm chooses the action with the maximal estimated action value, but with probability ϵ it selects an action at random. With a greedy policy (i.e., choosing the action leading to $\max_a Q(S_{i+1}, a)$), the two algorithms would match.

SARSA is usually faster to converge, while Q-learning is usually better at finding the best policy.

2.4.2 Reinforcement learning in software testing

Reinforcement learning has not yet been largely adopted in software testing research, especially in the CPS context. A summary of related work follows.

Ma et al. rely on reinforcement learning to perform model-based testing of a self-healing CPS [34]; Veanes et al., instead, perform model-based online testing of a simplified robot system [52]. Different from these two approaches, in COSMOS, we aim to test the CPS software without the aid of CPS models.

Yamagata et al. compare the performance of approximate methods (i.e., Asynchronous Advanced Actor Critic and Double Deep-Q Network), to address the robustness-guided falsification problem on CPS (i.e., to identify inputs that falsify a temporal property) [53]. Their results show that reinforcement learning outperforms baseline methods such as random sampling, cross entropy, and simulated annealing (the latter being a search-based method), thus further motivating our choice to investigate reinforcement learning in COSMOS. The main difference with COSMOS is that Yamagata et al. aim to verify temporal properties in a model, while in COSMOS, we aim to test an executable software component or system; software components and systems have a much larger input space than high-level models, which makes the testing process less scalable, thus requiring adequate solutions (e.g., approximate methods may be inapplicable).

Koren et al. demonstrated the feasibility of relying on reinforcement learning to test autonomous vehicles [32]; however, they rely on a simplified simulation software and case study.

Other software testing techniques relying on reinforcement learning target systems other than CPS. For example, Kim et al. demonstrated the feasibility of relying on reinforcement learning to achieve high structural coverage for simple programs [30]. Ahmad et al. rely on Q-learning for performance testing [7]. Adamo et al [5], Pan et al. [38], and Zheng et al [54] rely on Q-learning to perform GUI testing for Android and Web applications.

Based on the above, we conclude that COSMOS will be the first available toolset addressing the problem of effectively testing CPS through reinforcement learning.

3 COSMOS CPS Testing Automation

The COSMOS CPS Testing Automation Approach is implemented in the Test Execution Automator (TEA) component, which has been introduced in *COSMOS deliverable D2.1*. In this Chapter, we describe the methods that drive our approach (Section 3.1), the architecture of TEA (Section 3.2), and the end-user utilities that we provide (Section 3.3).

3.1 RL-based Testing Methodology

Our approach follows a workflow consisting of the following four stages:

1. the engineer defines the testing problem, which leads to the identification of a configuration for the reinforcement learning (RL) algorithm;
2. the RL model is trained;
3. the trained RL model is used to test the SUT;
4. the engineer analyzes the results; more precisely, she relies on the COSMOS tools to either inspect the reported failures individually or perform some aggregate analysis.

The workflow above can be instantiated into two distinct processes depending on the context of use, which are (1) *testing a software component from scratch* (e.g., the first version of the component), and (2) *testing an updated version of the software component based on the learned models*. These two processes are illustrated in Figures 4 and 5, respectively. The main difference between the two cases is that, when testing a software component from scratch (e.g., the first version of the component), the engineer needs to define the testing problem, whereas, when testing an updated version of the software, it is not necessary to redefine the testing problem and the RL model derived for the previous software version can be reused. In addition, when testing updated software versions, the aggregate analysis shall be followed by an analysis aiming to identify regression failures.

The following subsections describe the steps performed when testing a software component from scratch, followed by a detailed description of the activities performed when testing an updated software version. Please note that, since we perform black-box testing, we can rely on the same workflow when testing either a single software component or a whole software system. Hereafter, we simply refer to the *CPS software under test* as *SUT*.

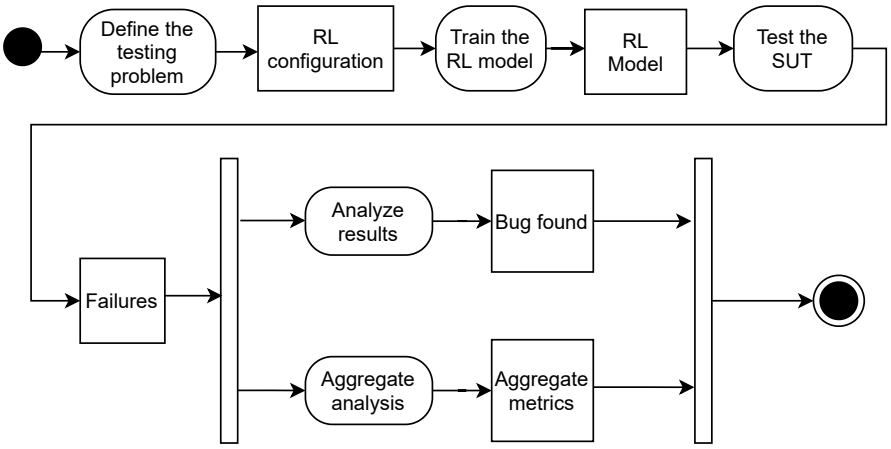


Figure 4: Testing a software component from scratch

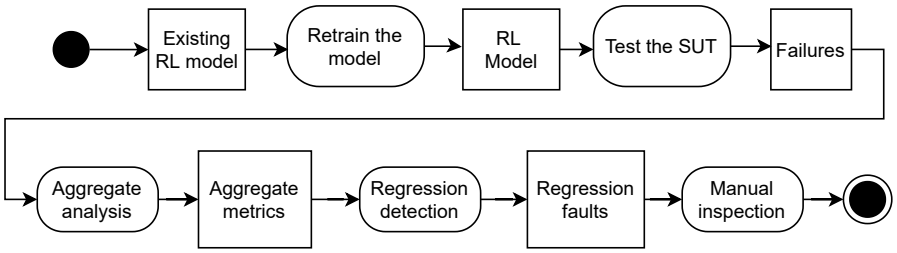


Figure 5: Testing an updated version of the software component

3.1.1 Define the testing problem

Based on the requirements under test, the engineer needs to define the *RL algorithm configuration* which consists of definitions for (a) the observation space, (b) the action space, (c) the initial state, (d) the reward function, (e) the RL algorithm to be used, (f) the execution budget, and (g) the simulator connector; we describe them in the following.

The *observation space* consists of the set of output variables provided by the simulator and the set of output variables provided by the SUT; more precisely, we shall focus on variables that provide information useful to determine the state of the environment and the system. In CPS software, such output variables might be either continuous or discrete. However, RL tabular algorithms such as Q-learning or SARSA can work only with a discrete representation of the observation space; indeed, each possible state of the system and the environment should be mapped to a cell of the Q-table. Consequently, the engineer needs to provide a discretization and normalization strategy; such strategy consists of a function that maps the domain of an output variable to a discrete value. For example, one common approach is to discretize continuous variables into a small number of buckets (each one represented by an integer). The discretization and normalization strategies defined by the engineer enable deriving a set of abstract states for the SUT and the environment; we use the term *system abstract state* to refer to a specific abstract state for the SUT and the environment (e.g. severe turbulence, mild headwind, and low altitude for the plane). Each system abstract state is univocally identified by a tuple that concatenates the discrete values obtained from each variable in the observation space; each tuple can be easily mapped to a cell of the Q-table.

To ensure scalability, the size of the observation space should be kept small (i.e., the number of distinct states for the system shall be low); indeed, the RL algorithm needs to bring the system into the different states and trigger different actions in order to learn which is the action that helps reach the testing goal. However, a very coarse discretization strategy may prevent the RL algorithm from identifying failures; for example, if we do not distinguish between different wind intensities we may not be able to determine that only strong headwind may cause the plane to lose altitude (e.g., because we do not reach a state in which the environment has strong wind). To reduce the size of the observation space, the engineer should also avoid those variables that are not related to the functionalities under test. For example, light intensity (e.g., day or night) is not relevant when testing the capability of an autopilot control software to keep the desired altitude; however, the same variable shall be considered when testing landing features that rely on runway lighting. In general, the engineer may need to identify a correct balance between scalability and effectiveness. We believe it to be feasible for engineers expert for the SUT domain, which should be the case for engineers involved in the SUT implementation; we leave the automated identification of an appropriate strategy to the coming months.

The *action space* consists of the actions that the RL agent shall take, that is, the inputs it shall provide to either the simulator (e.g., change wind speed) or the SUT (e.g., disable the *vertical speed hold* option in an airplane autopilot when testing for the *altitude hold* functionality). RL tabular methods only support a discrete action space; however, input variables for the SUT or the simulator may be continuous (e.g., wind speed in km/h). For this reason, it is necessary for the engineer to identify a set of *abstract actions* that the RL algorithm may select and the *mapping* between these abstract actions and the concrete values to be provided to the SUT and simulator. A common strategy for acting on a continuous input variable is to create a pair of abstract actions

which respectively increase or decrease the value of the variable by a constant amount. As with the observation space, the action space should be kept as small as possible.

The *initial state* of each episode simply consists of the initial assignments to input variables for the SUT and simulator. The initial state may either be fixed or vary between episodes (for example, each episode may start with a random initial state).

The *reward function* which examines the system state (i.e., the non discretized values assigned to the variables in the observation space) after every step, provides the reward signal for the RL algorithm, and recognizes termination conditions.

The *RL algorithm to be used* depends on the nature of the SUT and the testing problem. We currently provide implementations for Q-learning and SARSA, two popular tabular methods. Algorithms provided by external RL frameworks are also supported as described in Section 3.2.4.

The *execution budget* specifies the time to dedicate to training the RL model and testing the SUT; it is expressed as a maximum number of episodes during which testing will be conducted.

The *simulator connector* is a software component that enables communication between TEA and the simulator; connectors are described in Section 3.2.1. In our implementation, we provide two connectors (one for FlightGear and one for Pylot).

The engineer may define a distinct RL algorithm configuration for each requirement under test. For example, one RL algorithm configuration may aim to test the safety of an autopilot controller in realistic flight conditions, a slightly different configuration may be used to test its capability to reach a target altitude (e.g., initial conditions and reward vary), another one (e.g., different reward) may be used to test its capability to maintain a target altitude.

3.1.2 Train the RL model

During this stage, an RL model is trained by relying on the RL algorithm selected by the engineer. As described in Section 2.4, the RL algorithm selects an action to be performed, then it observes the result, in a series of *episodes*. An episode starts by resetting the simulator to its initial state S_0, as defined in Section 3.1.1. As previously mentioned, S_0 can either be fixed or variable; e.g., a random initial state can be chosen within predefined boundaries for every episode. In each step $i \geq 0$ of an episode, the RL algorithm proceeds as follows:

1. it retrieves the state S_i, based on the variables in the observation space;
2. it selects an action A_i and executes it;
3. after a predefined delay, it retrieves the new state S_{i+1};
4. it relies on the reward function to compute the reward R_{i+1} and verifies if we are in the presence of a termination condition, based on the new state S_{i+1};
5. it updates the action probabilities;
6. if a termination condition was recognized by the reward function, it logs information about the termination condition, terminates the episode, and starts a new episode.

The current version of TEA implements Q-learning and SARSA, which are two temporal difference algorithms, meaning that they learn by continuously updating their estimates of a value function from previous estimates (i.e., they *bootstrap*). Both Q-learning and SARSA learn by computing a function $Q(S, A)$, which estimates the future rewards if action A is taken when the system is at state S. They were described in Section 2.4.

Note that Q-learning and SARSA both choose an action based on an ϵ-greedy policy derived from their Q-tables; precisely, at every step, either $\mathrm{argmax}_a Q(S_i, a)$ is chosen with probability $1 - \epsilon$, or any other action is chosen at random with probability ϵ. The Q function is stored as a table with a row for every state and a column for every action. In TEA, a system abstract state is univocally identified by the combination of the values of every discretized and normalized variable.

The size of the Q-table is the product of the number of observed variables, the number of possible discretized values for each observed variable, and the number of actions. The larger the table, the more episodes are required in order to estimate the value of Q, for every state and every action. Therefore, if running many episodes is costly, the size of the table should be kept as small as possible. This can be done by limiting the number of observed variables, the granularity of the discretization of these properties, and the number of actions.

3.1.3 Test the SUT

After the training budget has been exhausted (i.e., after the maximum number of episodes has been reached), the trained model can be used to discover failures by executing the algorithm for a predefined number of further episodes. The difference between this stage and the previous one is that during this stage, exploration is entirely replaced with exploitation. In other words, the hyperparameter ϵ is set to 0, making the RL algorithm *greedy*.

When the initial state is initialized with random values, assuming that the RL model has appropriately learned how to make the SUT fail, the testing phase enables us to derive statistics about the observed failures (see Section 3.1.4). If the testing stage is omitted, the results (e.g., the failure rate) would be influenced by how easy failures are to be found (which depends on the specific RL algorithm chosen and may not be necessarily relevant to the engineer). Since the training phase also typically involves a large measure of exploitation (determined by the hyperparameter ϵ), the results would also be influenced by how early in the training process specific failures were found (which depends on random chance and might therefore randomly change between runs and between SUT versions).

3.1.4 Analyze results

During the training and testing stages, TEA collects *runtime data* that might be useful to debug the SUT; more precisely, it collects the initial state, the termination condition, the action that was taken at every step, the time at which that action was taken, the following state, and the reward.

Once testing is complete, two broad approaches to interpreting the results are possible: *manual inspection of individual failures* and *aggregate analysis*. The two approaches are compatible and simple tools are provided for both. The choice between the two can depend on how common failures are, and how significant each failure is. For example, in the presence of a few failures (e.g., less than ten), *manual inspection* is appropriate. To this end, we provide the tool *tea-replay* (see Section 3.3), which enables re-executing every episode exercised during RL training or SUT testing — a necessity to debug the SUT. However, with cyber-physical systems, failures are often inevitable given sufficient adverse conditions. In such cases, some *aggregate analysis of failures* may be necessary. To this end, we provide the tool *tea-analyze* (see Section 3.3), which computes three metrics:

- *Failure rate*. It is a simple aggregate metric that consists of the proportion of episodes ending with a failure.
- *Average episode length before a failure*. It reports the average number of steps taken before a failure is observed, in failing episodes. It is an indicator of how simple it is for the RL algorithm to make the SUT fail. It reflects the likelihood for a failure to be observed in reality (i.e., if a few steps are sufficient then it is very likely to be observed, if a long sequence of steps is needed then it is unlikely).

- *Adversity*. It is a cumulative measure of domain-specific adverse conditions. For example, it measures the cumulative amount of wind endured by the aircraft throughout an episode. Deviations in the minimum adversity which is necessary for a failure to occur would then indicate a regression fault. For instance, if in a new version of the SUT, a lower amount of wind is sufficient to cause an aircraft to crash, this likely indicates a regression fault.
  Adversity is defined as the vector $\left\{\int f(x_0)dt \quad \int f(x_1)dt \quad ... \quad \int f(x_{K-1})dt\right\}$ where x_k is the k-th input variable and K is the size of the observation space. The integrand function $f(x)$ can be x, x^2, or $|x|$; all of these are supported by *tea-analyze*. If an input variable can have both positive and negative values, these could cancel out in the computation of $\int xdt$. This effect could be appropriate according to the semantics of an input variable, for example if it is a measure of how much of a resource has been gained or consumed. In other cases it is inappropriate, for example in the case of wind which causes an aircraft to rotate around its longitudinal axis (i.e., wind affecting the aircraft's *roll*): in this case, the force of the wind is more important than the direction of rotation, and an even integrand function like $f(x) = x^2$ or $f(x) = |x|$ should be used instead. Using $f(x) = x^2$ amplifies the impact of outlying data points; if this is undesirable, $f(x) = |x|$ should be used.

3.1.5 Testing updated SUT versions

Testing updated SUT versions is essential in a CI/CD context. The TEA workflow for testing updated SUT versions presents two main differences with respect to testing the SUT from scratch, which are described below.

One difference is the availability of both an RL algorithm configuration (i.e., problem definition) and an already trained RL model. The RL model can either be erased after each change to the SUT, or reused between changes. The decision should be determined based on the characteristics of the specific testing problem, such as the cost of training, the scope of the changes, and the hyperparameter α (the learning rate). The optimal decision, which is beyond the scope of this document, is an avenue for future research.

The other main difference is the need for detecting regression faults (i.e., faults that appear after a change to the SUT or its environment), which we address through two separate strategies. Our first strategy for the identification of regression faults relies on aggregate analysis: if failures become more common after a change to the SUT, a regression is likely. The metrics discussed in Section 3.1.4 (e.g., failure rate and the adverse conditions) can be used for this purpose. Our second strategy for the identification of regression faults consists of automatically reproducing failures found on the latest SUT version on the previous version: if failures cannot be reproduced on the previous versions, they may be due to a regression. Finally, an opportunity for future development is the automated extraction of representative passing episodes, which could be used to build a regression testing suite to be executed before RL-based testing.

3.2 TEA Architecture

TEA consists of seven main components (i.e., Connectors, Properties, Actions, Environment, Model, Testing problem), which we describe below. In our implementation, we use the term property to refer to input or output variables for the simulator or the SUT (actually, we assume the simulator acts as a proxy for SUT variables). The relationships among them are illustrated in fig. 6. A detailed API reference can be found in appendix B.

3.2.1 Connectors

Connectors provide an interface between the simulator and TEA. A connector for the FlightGear flight simulator is provided with TEA; a prototype connector for the autonomous vehicle platform Pylot has also been developed. In order to use TEA with any other simulator, a connector needs to be implemented. Connectors handle communication with the simulator, including simulation errors; they can also handle the simulator startup and reset, as necessary.

Connectors shall implement a simple interface (defined in section B.2.1), which is built around the concept of *properties*. Properties represent any simulation attribute that can be queried and/or modified. Each property must be univocally identified by a string, and its value must be of one of the following data types: *None, bool, int, float, string*. Examples of properties for a flight simulator are the current altitude of the aircraft, the turbulence, and the rendering quality. Properties are used both to observe the state of the simulation for the purpose of learning, and as the object of actions.

Connectors are required to implement three methods: $get_property$, $set_property$, and $new_episode$. The last of these resets the simulator for a new episode. For more details, see section B.2.1.

3.2.2 Properties and Actions

Since properties typically need to be discretized and normalized before being used for learning, and actions can modify them in different ways, TEA provides two interfaces (one for properties in the observation space, one for actions) that facilitate the implementation of those standards procedures. Also, we provide several implementations of each of these interfaces which apply common strategies for handling observations and actions.

Implementations of the first interface (i.e., `ObservedProperty`, Section B.4.1), shall provide a method for normalizing the raw value obtained from the connector, as well as an imputed value to use when the property is not available. Several implementations are provided which apply common approaches for normalizing data; see Section B.4.2.

Actions are only required to implement a single method named *execute*. In practice, this will usually entail setting a property using the connector. Again, several implementations are provided for common approaches; see Section B.1.2.

3.2.3 Environment

The environment is an abstraction layer between the simulation and the machine learning model. It is modeled on OpenAI Gym environments[7], and has an analogous API. The environment handles obtaining properties using the connector, executing actions, calling the reward function provided by the engineer, and logging episodes and steps.

The environment class is documented in Section B.3. A fully compliant OpenAI Gym subclass is also provided; see Section B.3.1. It allows using any reinforcement learning algorithm compatible with OpenAI Gym, for instance those implemented by the stable-baselines project[8].

[7] See `https://github.com/openai/gym/blob/master/docs/api.md`

[8] Stable-baselines provides the implementation for a set of well-known RL algorithms, see `https://stable-baselines3.readthedocs.io/en/master/guide/algos.html`

3.2.4 Model

The model is responsible for the reinforcement learning implementation. TEA includes one model, `TabularModel` (see section B.7), which implements two widely known reinforcement learning algorithms: SARSA and Q-learning. A model can be trained, used for testing, loaded from a file, and saved to a file.

3.2.5 Testing problem

The testing problem is an interface that describes the configuration of a simulation-based test; it encompasses the entire problem definition described in Section 3.1.1, i.e. the observation space, action space, reward function, the RL algorithm (i.e., the model), and the connector. Every different configuration should be specified as a separate testing problem. Testing problems should implement the `TestingProblem` interface (see Section B.8). This allows them to be analyzed more easily by `tea-analyze` and `tea-replay` (see Section 3.3).

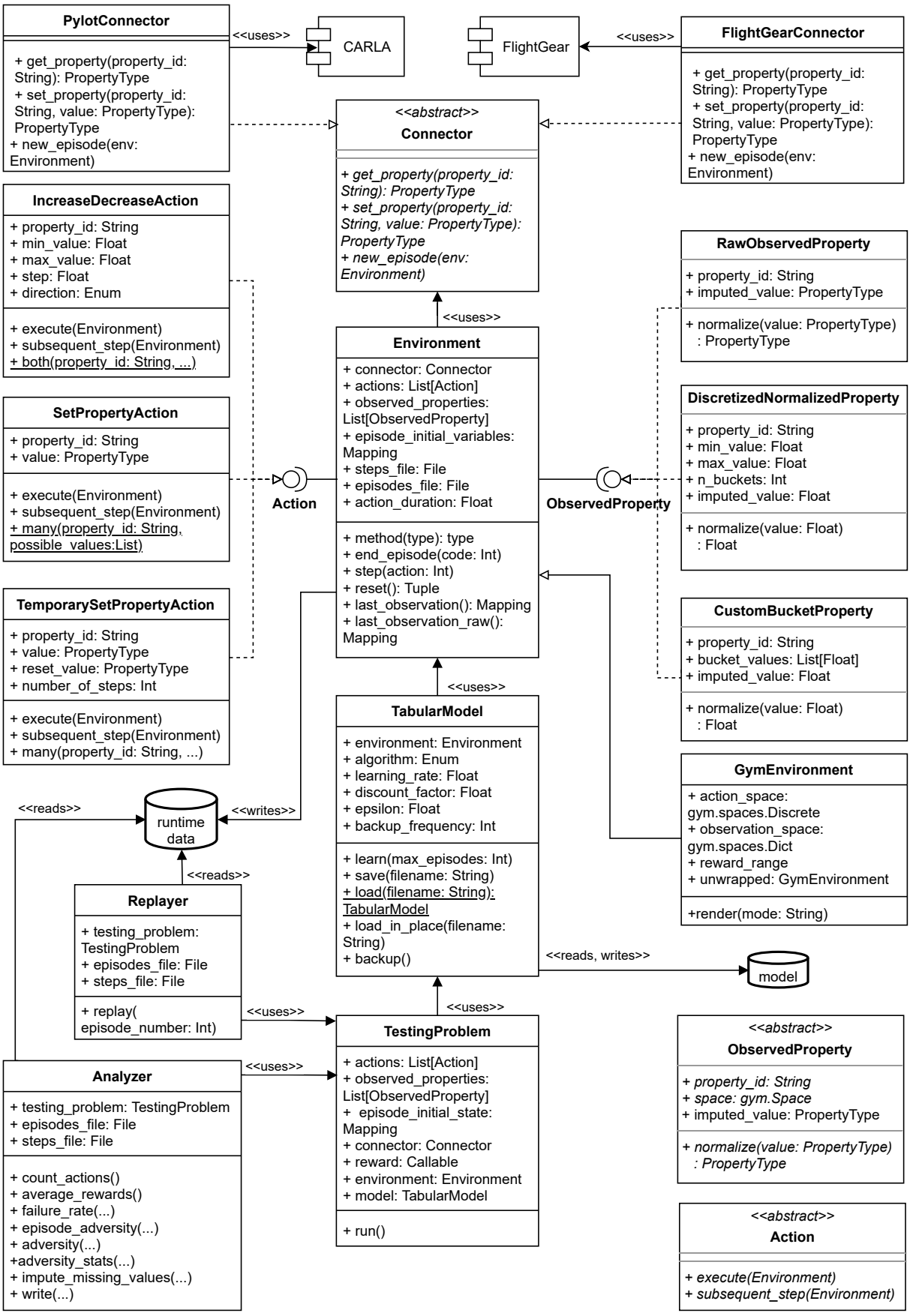


Figure 6: A UML class diagram showing the architecture of TEA

3.3 TEA Tools

The tools below have been developed to facilitate the testing process. Please note that all of these tools are in proof-of-concept status and are in need of further development and testing.

- TEA Dashboard is a graphical user interface for the TEA toolset. Currently, it only allows running simulators, tests, and other tools (such as the tools listed below), configuring how they should be run, and observing simulation attributes in real timee. However, it is planned to be extended with other features, such as observing the results of past tests.
- `tea-cli` is a command-line interface for TEA. It facilitates running simulators and tests automatically, e.g. as a hook in a continuous integration pipeline.
- `tea-analyze` is a tool for the aggregate analysis of the log files produced during testing. It reports the failure rate, adversity conditions, the average rewards for each action, and the number of time each action was performed. Its functionality is also exposed in the API in the Analyzer class.
- `tea-replay` reads the log files produced during testing and attempts to recreate an episode by performing the same actions at the same time. Its functionality is also exposed in the API in the Replayer class.

For details on how these tools are used, see appendix A.

4 Preliminary evaluation

We have performed a preliminary evaluation with two case studies focused on autonomous flight and autonomous driving, respectively. The former concern testing an autopilot software for FlightGear [19], a widely used open-source flight simulator. The latter concern testing the collision avoidance system of Pylot [24], which is an autonomous vehicle platform, on the CARLA simulator [14], which is the most used platform in software testing research [55]. The selected case study subjects are related to the case study subjects provided by COSMOS use case providers, specifically GMV and AICAS. Indeed, the use case provided by GMV concern software tested using the ATC-PIE simulator[9] within the FlightGear environment. AICAS, instead, provides a framework (i.e., Jamaica EDP) for integrating and testing automotive components; it can be used with different simulation environments including Carla. Additional information about the integration between Jamaica EDP and Simulator environments are provided in COSMOS deliverable D2.2.

Our future work concerns performing an extensive empirical evaluation of the approach including its application to the COSMOS use cases. To this end, we aim to apply it not only to the above-mentioned GMV and AICAS cases but also to the UNPARALLEL Utility Management simulated environment. Finally, the COSMOS CPS testing approach might also be applied to the Railway system provided by QMedia, which relies on the PikeOS network simulation features; however, the use case is mostly focused on security rather than functional and functional-safety testing.

4.1 Case study 1: Autonomous flight

Our first case study concerns FlightGear [19], a widely used open-source flight simulator that is used for academic research and education (e.g., train pilots). We rely on COSMOS to test IT-AUTOFLIGHT [20], an autonomous flight controller for FlightGear. Figure 7 shows the IT-AUTOFLIGHT interface within the FlightGear simulator; IT-AUTOFLIGHT resembles the interface of an autopilot component that can be interacted with by a human end-user (in our context, we can simulate such interaction to be performed by the RL agent, if needed, through the Telnet interface of FlightGear).

In this case study, we test the requirement *the airplane shall reach a target altitude at a minimum average speed of* $1ft \cdot s^{-1}$ *(i.e., one foot per second)*. In our test scenario, IT-AUTOFLIGHT is configured to bring an aircraft to a target altitude from a lower starting altitude. The RL agent seeks to hinder this, either by crashing the aircraft or by slowing down its ascent. The RL agent achieves this goal by controlling some aspects of the environment, namely the wind speed and turbulence.

4.1.1 Problem Definition

Simulation variables

FlightGear has thousands of input and output variables, the values of which can be retrieved and set during the simulation via a Telnet interface. These variables relate to various attributes of the simulated aircraft, such as its position, velocity, and instrument readings, as well as the environment, e.g. atmospheric conditions and other weather variables. To perform the case study, we first identified a subset of variables that relate to factors that may affect an aircraft's capability of flight without crashing (the complete set is reported in appendix C and can be used as a reference to test other requirements of IT-AUTOFLIGHT); then, we selected a subset of variables that concern environment elements that may affect the requirement under test. For our case study, we considered the following variables:

[9] https://wiki.flightgear.org/ATC-pie

Figure 7: IT-AUTOFLIGHT interface within FlightGear

- *Pressure altitude*, as measured by an altimeter based on the pressure of the surrounding air and relative to the mean sea level;
- *Absolute altitude*, also known as *above-ground level (AGL) altitude*, as measured by a ground radar and relative to the ground beneath the aircraft;
- *Wind speed* in each of the three axes, with positive values indicating wind in one direction along the axis and negative values indicating wind in the opposite direction along the same axis;
- *Turbulence*, on a scale from 0 (none) to 1 (severe).

Variables such as the attitude of the aircraft (yaw, pitch, and roll) were excluded because they are dependent on other variables, such as the wind speed. Since IT-AUTOFLIGHT only relies on instrument readings and not on visual input, variables with mostly or entirely visual impact such as visibility were disregarded.

Data abstraction for observation space and action space

While no normalization or discretization is performed in computing the reward signal, variables forming the action space and the observation space are normalized and discretized into buckets. Specifically, each variable is manually assigned a range which is then divided into a number of intervals called *buckets*, corresponding to equally spaced *nominal values* of the variable. A continuous variable is then discretized by replacing it with the number of the bucket corresponding to the closest nominal value. Values outside the assigned range for a variable are respectively placed in the lowest or highest bucket. Therefore, although buckets represent equally-spaced values, the first and last bucket correspond to potentially infinite intervals for variables with theoretically infinite ranges, such as the wind speed. This normalization strategy reflects the observation that the ranges of the variables in question, although theoretically infinite, are in practice quite limited, as in the case of wind sped, or that no distinction is necessary among values outside the chosen range, as in the case of AGL altitude (since, for the purpose of this case study, the altitude of the aircraft matters little so long as it is not close to the ground).

Table 2 details how the variables considered for the observation and action space had been discretized. The observation space in this case study consists of five variables: turbulence, discretized into four buckets; AGL altitude, discretized into three buckets; and each of the three components (axes) of wind speed, discretized into seven buckets; Four of these variables, namely turbulence and the three components of wind speed are also the object of all actions. The action space consists of a pair of actions for each of these variables, where one

action increases its value, and the other decreases it, within the boundaries of the chosen range for the variable in question (i.e., within min/max in Table 2). These actions always modify the variable by a constant amount, equal to the width of the buckets chosen for that variable (i.e., each action makes the variable value to jump from one bucket to another).

Variable	Minimum	Maximum	Number of buckets
Turbulence	0	1	4
AGL altitude	900	2700	3
Downward wind	$-13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(-45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	$13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	7
Eastward wind	$-13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(-45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	$13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	7
Northward wind	$-13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(-45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	$13.716\ \mathrm{m}\cdot\mathrm{s}^{-1}$ $(45\ \mathrm{ft}\cdot\mathrm{s}^{-1})$	7

Table 2: Range and discretization granularity of each variable in the observation space

Initial state and termination conditions

Every episode starts with a random initial state, that is, a uniformly chosen value for each of the variables forming the action space within the chosen range for that variable. For this case study, we kept the initial and target altitude set to 6000 and 10000 feet, respectively. Randomization of the initial state is important during the testing phase, when the RL agent is greedy (i.e., purely exploitative), in order to avoid repetitive episodes. It also allows detecting new failures by exploring values of the variables that would otherwise never be reached (as actions would only ever jump from one nominal value to the next, passing over all values between them). This is because the increase/decrease action pairs used in this case study always modify the variable they control by a constant, predefined amount.

An episode terminates with a failure (i.e., the test case fails) in two distinct cases: (1) if a crash is detected, or (2) if the target altitude is reached, but the overall ascent is slow. The overall ascent is considered slow when the average rate of ascent from the start of the episode is lower than v_{min}, a parameter we set at $0.3048\ \mathrm{m}\cdot\mathrm{s}^{-1}(1\ \mathrm{ft}\cdot\mathrm{s}^{-1})$, i.e., 50% slower than what it would have been if there were no wind and no turbulence. An episode is successfully concluded (i.e., the test case passes) if the target altitude is reached with an overall rate of ascent above v_{min}. Finally, the episode might be terminated ignoring its outcome and its effect on the RL model if FlightGear becomes unreachable (it may happen during long test execution runs).

Reward function

The reward function R_i at step $i \geq 1$ is defined as follows:

$$R_i = \begin{cases} 0 & i = 1 \vee a_i \geq 0 \\ \frac{a_i}{a_{max}} & i \geq 2, \quad a_{max} < a_i < 0 \\ 1 & i \geq 2, \quad a_i \leq a_{max} \end{cases} \tag{3}$$

where a_i is the *vertical acceleration* of the aircraft at step $i \geq 2$ and a_{max} is a parameter which we set at $-3.048\ \mathrm{m}\cdot\mathrm{s}^{-2}(-10\ \mathrm{ft}\cdot\mathrm{s}^{-2})$. Note that a negative vertical acceleration indicates either decelerating ascent or accelerating descent.

a_i is calculated based on the previous altitude observations as follows:

$$a_i = \frac{v_i - v_{i-1}}{t_i - t_{i-1}} \tag{4}$$

$$v_i = \frac{h_i - h_{i-1}}{t_i - t_{i-1}} \tag{5}$$

where t_i, h_i, and v_i respectively represent the time, the pressure altitude of the aircraft, and its vertical velocity at step i. Thus the reward is 0 if the vertical acceleration following an action is positive (i.e., normal operation for the aircraft), and rises if the vertical acceleration is negative, up to a maximum of 1, reached for $a = a_{max}$. After that, it stops rising.

4.1.2 Experiment

A model of the RL problem described above was trained for 1000 episodes using the SARSA algorithm, and then used to test the SUT for 250 episodes. 163 of the 250 testing episodes, i.e. 62.5 %, failed, while the remaining 87 episodes passed. The high proportion of testing episodes leading to a failure highlight the potential effectiveness of the approach.

On a laptop with 2.7 GHz Quad-Core Intel Core i7 and 16GB RAM, training required 80 hours while testing required 36 hours. Please note that testing can be easily parallelized (e.g., so that it can be executed overnight) while long training may not be required if trained models are reused from previous versions. Also, a much lower number of episodes might be sufficient to detect failures. Future work includes the identification of such optimizations for better integration within a CI/CD context.

Concerning aggregate analysis, failing testing episodes had an average length of 93 steps, compared to 285 steps for passing ones. Table 3 shows the adversity, as defined in section 3.1.4, which was measured for each of the variables in the observation space, with the exception of altitude. $f(x) = x^2$ was used as an integrand. The adversity for every variable was compared across all testing episodes which ended in failure, resulting in the minimum, mean, maximum, and standard deviation values in table 3. Note that wind speeds are given along a geometrical axis, with the sign indicating the direction along the axis (e.g. a negative downward wind indicates an upward wind). Results show that failures might be observed even in the absence of turbulence (e.g., in the case of high wind speed); for the future, we aim to include analyses that enable us to derive dependencies across multiple variables (e.g., decision trees) thus better characterizing the unsafe input space.

The average episode length and adversity calculations were produced by *tea-analyze* (see section 3.3) and can be used to detect regression faults by comparing them to the results for previous SUT versions.

Variable	Minimum	Mean	Maximum	Standard deviation
Turbulence	0	134	2246	241
Downward wind	23016	385080	22445029	1747091
Eastward wind	9052	293518	15360324	1217504
Northward wind	2038	323743	16233721	1279809

Table 3: Adversity of each type across testing episodes ending in failure

4.2 Case study 2: Autonomous driving

This case study concerns the CARLA driving simulator [14], in conjunction with Pylot [24], an autonomous vehicle platform for developing and testing autonomous vehicle components on the CARLA simulator. Pylot also supports physical vehicles, potentially enabling HIL testing. Figure 8 shows the architecture of Pylot; please note that it consists of several DNNs each addressing one specific driving task.

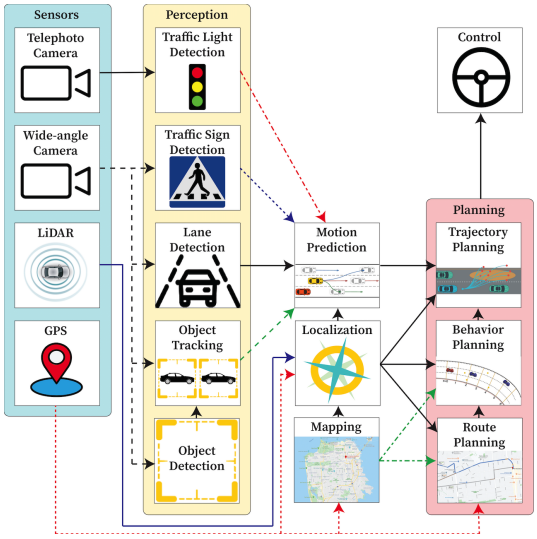


Figure 8: Pylot architecture.

For our case study, we test the collision avoidance system of Pylot version 0.3.2 (Mar 31, 2021). In our test scenario, an autonomous vehicle (hereafter referred to as the *ego vehicle*) is controlled by Pylot, while the RL agent controls the vehicle in front of it with the objective of provoking a collision between the two vehicles.

4.2.1 Problem definition

Various simulation variables can be controlled during the simulations, including aspects of the environment (such as the time or day and the corresponding lighting conditions) and the vehicle in front. For the purposes of the case study presented here, the following variables were taken into account:

- the distance between the ego vehicle and the vehicle in front of it, normalized into the range $[0, 1]$ and discretized into 10 buckets;
- the speed difference between the two vehicles, normalized into the range $[0, 10]$ and discretized into 1000 buckets;
- the throttle of the vehicle in front, normalized into the range $[0, 1]$ and discretized into 5 buckets;
- the steering of the vehicle in front, normalized into the range $[-0.01, 0.01]$ and discretized into 3 buckets.

The observation space consists of all of the variables above, while the action space only affects the last two. Analogously to the previous case study, the action space consists of a pair of actions for each of the two variables (i.e. the throttle and steering of the vehicle in front), where one action increases the value of the variable, and the other decreases it. Again, these actions always change the variable by a constant amount, equal to the width of the buckets chosen for that variable, and within the range of that variable.

Episodes are deemed to fail when a collision with the vehicle in front occurs. Three other non-failing termination conditions are defined: when the ego vehicle reaches its destination; when its movement is blocked due to the positions of other vehicles; and when a maximum number of actions is reached (this termination condition was added in order to provide a maximum time limit).

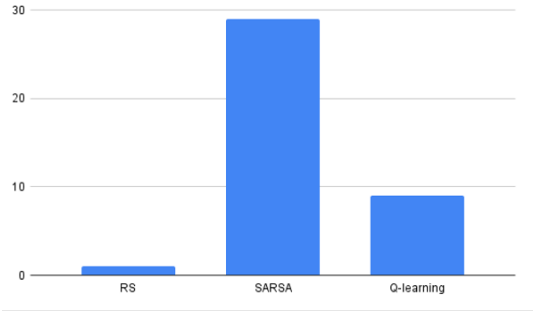


Figure 9: Number of failures found using SARSA and Q-learning (baseline: random search)

The reward function is defined as follows:

$$R_i = \begin{cases} \frac{1}{d} & d > 0 \\ 1000 & d = 0 \end{cases} \tag{6}$$

where d is the distance between the two vehicles at step i. Intuitively, the reward increase when the two vehicles get closer (it means that the RL agent successfully make the autonomous car less safe). The reward function reaches its maximum value when the two vehicles crash.

4.2.2 Experiment

A model of the RL problem described above was trained for 200 episodes, and then used to test the SUT for 50 episodes. This was repeated for Q-learning, SARSA, and random search as a baseline. During the 50 testing episodes, the model trained using SARSA found 29 failures (i.e., collisions between the two vehicles), while the one trained using Q-learning found 9; random search only found 1 (see Figure 9). These preliminary results show that RL is indeed an effective solution for the identification of failures (random search has a much lower effectiveness). Also, SARSA outperforms Q-Learning; such difference is likely due to Q-Learning's need of a larger budget for training. In a CI/CD context, where testing might need to be executed frequently, the testing budget is clearly an important factor for choosing the algorithm to adopt. For this reason, we believe SARSA may be a better candidate in this context.

Figure 10 shows a subset of the frames recorded in the video capture for one of our failing episodes; the failure occurs when the front vehicles slowly goes on its right (somehow, the DNNs do not realize that the front vehicle cannot cross the guardrail).

On a system with Ubuntu 18.04 running on Intel i9-9900K CPU with RTX 2080 Ti (11 GB) and 32 GB memory the training of the RL agent took 25 hours, testing 6 hours. As for the FlightGear case, our future work includes the identification of solutions to execute testing within the budget required in a CI/CD context (please note that, in this case, testing can already be performed overnight).

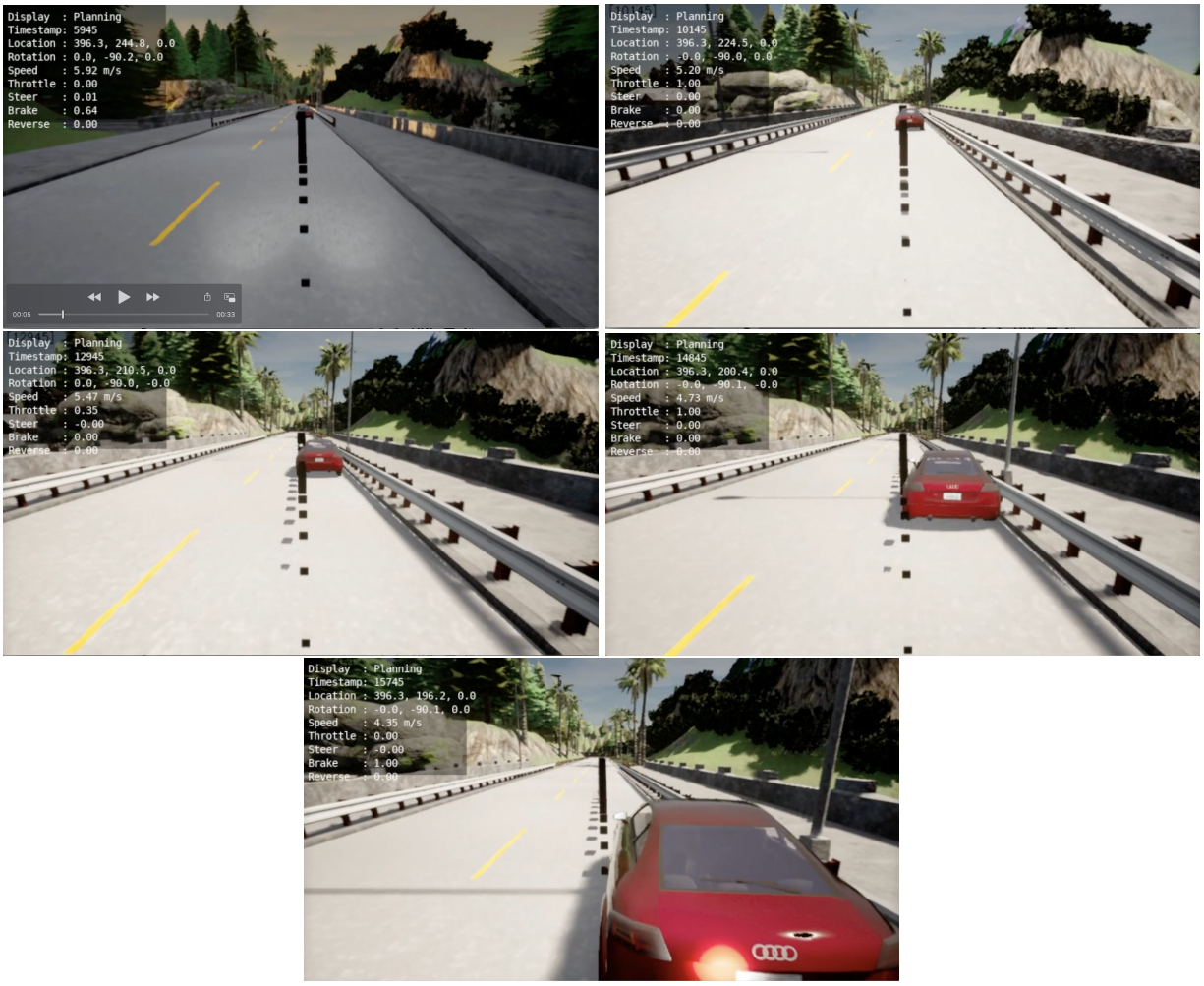


Figure 10: Failure detected by the COSMOS CPS Testing toolset applied to Pylot. We report a subset of the frames recorded in the video capture for the failing episode.

5 Conclusion

In this deliverable, we presented the preliminary COSMOS solutions for the automated testing of software running on CPS. Following trends in related work, we rely on a black-box approach that can be applied to software components or whole software systems. Our approach relies on simulators but does not require models of the software under test (SUT) or its environment. To maximize testing effectiveness, we rely on reinforcement learning (RL), an AI solution to train an agent to achieve a goal. In our context, the goal of the agent is to make the SUT fail. We believe RL to be more effective than state-of-the-art solutions for software testing (e.g., evolutionary algorithms) because it allows simulating an agent (e.g., an environmental agent, or an actor of the system) that reacts to the actions taken by the SUT in a timely way in order to make it fail.

Our preliminary evaluation conducted with two case study subjects has shown that our RL-based approach performs better than baseline random search in detecting failures; furthermore, our aggregate analysis has shown potential in helping engineers draw conclusions from test results. Finally, the SARSA RL algorithm has shown to be more effective than Q-Learning, in our context.

Our future work concerns performing an extensive empirical evaluation of the approach including its application to the COSMOS use cases. In particular we aim to consider the case study subjects provided by COSMOS use case providers that work with related simulators (i.e., specifically GMV and AICAS); we also aim to apply it to the UNPARALLEL Utility case.

A Tutorial

This section guides you through the code of the autonomous flight case study (see section 4.1). The full code of the example can be found in the file `flightgear_case_study.py`.

Example prerequisites

FlightGear is a widely-used open-source flight simulator. It supports getting and setting properties via a Telnet interface. Follow the following steps in order to install FlightGear, as well as the aircraft and the autopilot used in this example.

1. Download and install FlightGeear from `https://www.flightgear.org/download/`
2. Download the MQ-9 Reaper UAV from `http://mirrors.ibiblio.org/flightgear/ftp/Aircraft/MQ-9.zip`
3. Download IT-AUTOFLIGHT from `https://github.com/Octal450/IT-AUTOFLIGHT/releases/tag/4.0.6`
4. Follow the installation instructions on `https://wiki.flightgear.org/IT-AUTOFLIGHT`
5. Run FlightGear with the parameters as in `keepRunningFG.sh`. Manually edit the `MQ-9-JSBSim2-set.xml` file.

You can now run the example: `python3 flightgear_case_study.py`

Example

This example follows the general steps presented in section 3. First, create a sub-class of TestingProblem. In the constructor, start defining the observation space as follows:

```
class FlightGearCaseStudy(TestingProblem):
    def __init__(self):
        self.actions = [
            IncreaseDecreaseAction.both(
                '/environment/config/presets/turbulence-magnitude-norm',
                min_value=0.00,
                max_value=1.00,
                step=(1 / 3)
            ),
        [...]
```

In the snippet above, we add *two actions* acting on turbulence; the first one increases it and the second one decreases it with respect to its value at the time the action is executed. Both do so by increments or decrements of 1/3, within the interval [0.0; 1.0]. Three analogous pairs of actions are then added in a portion of code that is not reproduced here, corresponding to wind speed from each of the three directions. Note that IncreaseDecreaseAction.both returns a list of two actions within the bigger list of all actions. This is not a problem because Environment reads the nested list as if it were flat.

Now we need to define observed properties:

```
self.observed_properties = [DiscretizedNormalizedProperty('/environment/
↪config/presets/turbulence-magnitude-norm', 0.0, 1.0, 4),
[...]
```

This time, we're adding the same property as an observation property. By using DiscretizedNormalizedProperty with the parameters above, we are specifying that the raw value of the property should first be normalized in the interval [0.0; 1.0], and then discretized in 4 buckets within this interval. Since the buckets are equally big, this means that after normalization, the property will have one of the following values: 0.0, 1/3, 2/3, 1.

After adding the other observed property analogously, we need to define an initial state for every episode, in the form of a function `generate_initial_state` which returns a mapping of properties to the values they should be assigned at the beginning of every episode:

```
self.episode_initial_state = generate_initial_state
```

This function is defined as follows:

```
def generate_initial_state():
    return {
        '/environment/config/presets/turbulence-magnitude-norm-override
↪': True,
        '/environment/config/presets/turbulence-magnitude-norm': random.
↪uniform(0.0, 1.0),
        '/fdm/jsbsim/atmosphere/gust-down-fps': random.uniform(-max_wind_
↪speed, max_wind_speed),
        '/fdm/jsbsim/atmosphere/gust-east-fps': random.uniform(-max_wind_
↪speed, max_wind_speed),
        '/fdm/jsbsim/atmosphere/gust-north-fps': random.uniform(-max_
↪wind_speed, max_wind_speed),
        '/it-autoflight/input/ap1': True,
        '/it-autoflight/input/fd1': True,
        '/it-autoflight/input/athr': True,
        '/sim/hud/visibility[1]': True
    }
```

The first property in the returned dictionary enables setting the turbulence using the second property in the dictionary. The following three set the initial wind speed to random in each direction. This also allows the RL agent to explore some values which would otherwise always be skipped due to the constant step of IncreaseDecreaseAction. The following two variables enable the autopilot and "Flight director".

The next step is initializing a connector, in this case a `FlightGearConnector`:

```
self.connector = FlightGearConnector(
            'localhost',
            port=9000,
            raise_exceptions=False,
            on_error=self.on_flightgear_error
        )
```

`self.on_flightgear_error` does the following:

```
def on_flightgear_error(self):
    logging.warning('FlightGear error. Sleeping for 15s...')
```

(continues on next page)

(continued from previous page)

```
    time.sleep(15)
    self.environment.end_episode(EpisodeCode.FlightGearUnreachable)
```

Now, we instantiate the environment:

```
self.environment = Environment(
    self.connector,
    self.actions,
    self.observed_properties,
    reward,
    self.episode_initial_state,
    open('episodes.csv', 'a', buffering=1),
    open('steps.csv', 'a'),
)
```

The last two arguments are the files where the steps and episodes respectively will be logged. `buffering=1` configures the file object to flush to disk after every line, since episodes are typically quite long. Steps are much shorter, so this is not necessary and could lead to excessive numbers of writes.

The reward function is quite long and intricate, so it is not reproduced here. In essence, it gives a high reward (up to a maximum of 1.0) when the aircraft is falling rapidly, a low reward if it is falling slowly, and a zero reward if it is rising or keeping its altitude. It also terminates the episode with a positive end-of-episode code if the aircraft has reached its target altitude or a technical error has occurred (e.g. FlightGear has crashed), and records a failure (signalled by a negative end-of-episode code) if the aircraft has crashed or if it has reached its target altitude too slowly.

Finally, we need to instantiate the model:

```
self.model = TabularModel(self.environment,
                                  RLAlgorithm.SARSA,
                                  backup_frequency=5)
```

and implement the `run` method:

```
def run(self):
    logging.basicConfig(format='%(levelname)s: %(message)s',␣
↪level=logging.DEBUG)
    self.model.learn(1000)
    self.model.save()
```

Note that FlightGearConnector assumes that FlightGear is already running, and will not start it if it's not. Also, when the aircraft crashes, FlightGear also tends to crash, so it is useful to run a script that keeps running it (like `keepRunningFG.sh`, which is included). Worse, sometimes it freezes and starts endlessly consuming memory until, eventually, all system memory is over. For this reason, it might be useful to limit the memory it can use, e.g. using `ulimit`.

As mentioned above, the full code of this example is contained in the file `flightgear_case_study.py`.

B API Reference

B.1 Actions

B.1.1 Interface

class Action

Bases: abc.ABC

This class represents an action.

TEA Actions typically act on properties, but they don't have to.

Note that a (shallow) copy of the action object is created before it is executed. This is done using copy.copy; if you want to customise this behaviour, override the __copy__ method. For more information, see https://docs.python.org/3/library/copy.html One copy is created every time an action is executed; subsequent_step is then called on the same instance at every step so long as it keeps returning True. Neither the execute nor the subsequent_step methods are ever called on the original instances passed to the environment object.

abstract execute(*env: Environment*) → None

Called once, when the action is executed.

subsequent_step(*env: Environment*) → bool

After execute is called, this method is called on every step so long as it keeps returning True. Once it returns False, the method will not be called again and the Action copy will not be used anymore by the environment.

B.1.2 Implementations

class IncreaseDecreaseDirection(*value*)

Bases: enum.Enum

Specifies which way an IncreaseDecreaseAction will go.

class IncreaseDecreaseAction(*property_id: str*, *min_value: float*, *max_value: float*, *step: float*, *direction:* actions.IncreaseDecreaseDirection)

Bases: *actions.Action*

An action that allows increasing/decreasing a property within a range

This action will increase or decrease the property it is bound to, with respect to its value at the time when the action is executed.

Parameters

- **property_id** – The identifier of the property to act on
- **min_value** – The minimum value the property is allowed to be assigned to
- **max_value** – The maximum value the property is allowed to be assigned to
- **step** – The amount the property will be increased/decreased by at a time
- **direction** – The direction of the change.

static both(*property_id: str*, *min_value: float*, *max_value: float*, *step: float*) → List[*actions.IncreaseDecreaseAction*]

Generate two actions of this type, one for each direction.

Parameters

- **property_id** – The identifier of the property to act on
- **min_value** – The minimum value the property is allowed to be assigned to

- **max_value** – The maximum value the property is allowed to be assigned to
- **step** – The amount the property will be increased/decreased by at a time

class SetPropertyAction(*property_id: str*, *value: Any*)

Bases: *actions.Action*

An action that sets a property to a constant value

Parameters

- **property_id** – The identifier of the property to act on
- **value** – The value which will be assigned to the property

static many(*property_id: str*, *possible_values: List*) → List[*actions.SetPropertyAction*]

Generate multiple actions of this type.

Parameters

- **property_id** – The identifier of the property to act on
- **possible_values** – A list of values the property can be assigned to; a separate action will be created for each

class TemporarySetPropertyAction(*property_id: str*, *value: Any*, *reset_value: Any*, *number_of_steps: int = 1*)

Bases: *actions.Action*

An action that temporarily sets a property to a constant value

Parameters

- **property_id** – The identifier of the property to act on
- **value** – The value which will be assigned to the property
- **reset_value** – The value to which the property will be reset after the set number of steps has elapsed
- **number_of_steps** – The number of steps before the property is reset

static many(*property_id: str*, *reset_value: Any*, *possible_values: List*, *number_of_steps: int = 1*) → List[*actions.TemporarySetPropertyAction*]

Generate multiple actions of this type.

Parameters

- **property_id** – The identifier of the property to act on
- **reset_value** – The value to which the property will be reset after the set number of steps has elapsed
- **possible_values** – A list of values the property can be assigned to; a separate action will be created for each
- **number_of_steps** – The number of steps before the property is reset

B.2 Connectors

B.2.1 Interface

class Connector

Bases: abc.ABC

An interface between TEA and a simulator.

This interface is used by TEA to communicate with the simulator. It can be implemented for any simulator which allows for relevant properties to be get and set externally during runtime. Properties must be uniquely identifiable using string identifiers; their type must be bool, int, float, or str.

abstract get_property(*property_id: str*, *default_value: Union[None, bool, int, float, str] = None*) → Union[None, bool, int, float, str]

This method must obtain the value of the property identified by property_id. If that property is unavailable, it must return default_value.

abstract set_property(*property_id: str*, *value: Union[None, bool, int, float, str]*) → None

This method must set the property identified by property_id to value.

abstract new_episode(*env:* environment.Environment) → None

This method is called by TEA every time a new episode is started.

get_properties(*property_ids: Sequence[str]*, *default_value: Union[None, bool, int, float, str] = None*) → Dict[str, Union[None, bool, int, float, str]]

Returns a mapping of several properties at once.

The default implementation of this method simply calls get_property as many times as necessary. You only need to override it if it can be implemented more efficiently.

set_properties(*values: Dict[str, Union[None, bool, int, float, str]]*) → None

Set several properties at once.

The default implementation of this method simply calls set_property as many times as necessary. You only need to override it if it can be implemented more efficiently.

exception ConnectorError

Bases: Exception

This exception must be raised for any errors related to the simulator, including connectivity errors and simulator errors.

B.2.2 Implementations

class FlightGearConnector(*host: str = 'localhost'*, *port: int = 5500*, *raise_exceptions: bool = False*, *on_error: Optional[Callable[[], None]] = None*, *number_of_retries: int = 3*, *refuel_interval: int = 100*, *refuel_amount_kg: float = 170.0*)

Bases: *connector.Connector*

A TEA connector to the FlightGear flight simulator

This connector controls FlightGear using the latter's Telnet interface. Note that it does not start or kill the FlightGear process; that task is left to the user to allow for maximum flexibility.

Parameters

- **host** – The name of the host running FlightGear
- **port** – The number of the port FlightGear is listening to using Telnet
- **raise_exceptions** – Whether exceptions should be raised on errors

- **on_error** – Called whenever an error occurs
- **number_of_retries** – How many times operations will be retried on error. If *raise_exceptions* is True, this parameter is ignored.
- **refuel_interval** – The number of operations after which the aircraft will be refuelled. If set to None, it will not be automatically refuelled.
- **refuel_amount_kg** – The amount of fuel (in kg) by which the aircraft will be refuelled.

class PylotConnector(*property_index: List[str]*, *pylot_workspace: str = '/home/erdos/workspace/'*, *local_workspace: str = './pylot-execution-data'*)

Bases: *connector.Connector*

A TEA connector to a Pylot/CARLA Docker container.

Parameters

- **property_index** – A list of the properties that form the simulation state, in the same order. These strings be used as property_id's.
- **pylot_workspace** – The *workspace* directory on the Pylot Docker container
- **local_workspace** – A directory on the machine where this module is executed, used for storing execution data locally

get_property(*property_id: str*, *default_value: Union[None, bool, int, float, str] = None*) → Union[None, bool, int, float, str]

Calls *get_properties()* and returns the value of *property_id*.

get_properties(*property_ids: Sequence[str]*, *default_value: Union[None, bool, int, float, str] = None*) → Dict[str, Union[None, bool, int, float, str]]

Fetches the simulation state and extracts the properties according to *property_index*.

set_property(*property_id: str*, *value: str*) → None

Sets the next command to be executed.

Currently the only supported property is *command*.

new_episode(*env:* environment.Environment) → None

Resets the simulation for a new episode.

B.3 Environment

class Environment(*connector:* connector.Connector, *actions: List[*actions.Action*]*, *observed_properties: List[*properties.ObservedProperty*]*, *reward: Callable[[*environment.Environment*], float]*, *episode_initial_state: Union[Dict[str, Union[None, bool, int, float, str]], Callable[[], Dict[str, Union[None, bool, int, float, str]]]]*, *episodes_file: Optional[TextIO] = None*, *steps_file: Optional[TextIO] = None*, *action_duration: float = 0.0*)

Bases: `object`

An abstraction of a simulated environment

The environment is an abstraction layer between the simulation and the machine learning model. It is modeled on OpenAI Gym environments[1], and has an analogous API. The environment handles obtaining properties using the connector, executing actions, calling the reward function provided by the engineer, and logging episodes and steps.

This class implements an API analogous to the one in OpenAI Gym[1]. However, if strict compliance with that API is necessary, use *GymEnvironment*.

[1] https://github.com/openai/gym/blob/master/docs/api.md

Parameters

- **connector** – The connector used to communicate with the simulator
- **actions** – A list of possible actions which will form the action space. This list will be flattened using *flatten()*, so it can contain lists or tuples of actions.
- **observed_properties** – The list of properties which will form the observation space. The order of properties will be preserved in the observation space mappings returned by *step()*, *reset()*, *last_observation()*, and *last_observation_raw()*.
- **reward** – The reward function which will be called after every step. An instance of this class will be passed as a single argument; the function should return the reward as a float. Besides serving as a reward signal, it can also recognise termination conditions and call *end_episode()*.
- **episode_initial_state** – Either a dictionary that maps property identifiers to the values these properties should take, or a callable which returns such a dictionary (without taking any arguments). This dictionary will be passed to `connectors.Connector.set_properties()` at the beginning of every new episode. For instance, it can be used in order to initialize every episode with a random state.
- **episodes_file** – A file-like object where information about every episode will be written. If None, nothing will be writen.
- **steps_file** – A file-like object where information about every step will be written. If None, nothing will be writen.
- **action_duration** – How long to wait at the end of every step. Note that if communication is slow, it may not be necessary to wait at all.

end_episode(*code: int*, *message: str = ''*) → None

End the current episode after the next step.

Parameters

- **code** – An integer code corresponding to the termination condition of the episode (analogous to a return code)
- **message** – A more detailed description of the termination condition (i.e., what caused the end of the episode).

last_observation() → OrderedDict[str, Union[None, bool, int, float, str]]
Return the (normalized) state of the observation space following the last action.

last_observation_raw() → OrderedDict[str, Union[None, bool, int, float, str]]
Return the (non-normalized) state of the observation space following the last action

reset(*episode_initial_state: Optional[Dict[str, Union[None, bool, int, float, str]]] = None*)
Reset the Connector for a new episode
Parameters **episode_initial_state** – The initial state of the new episode. If omitted, the initial state provided in the constructor will be used.

Returns The state of the observation space after the new episode has started.

step(*action_index: int*) → Tuple[OrderedDict[str, Union[None, bool, int, float, str]], float, bool, dict]
Executes *self.actions[action_index]*.
Parameters **action_index** – The index of the action to execute.

Returns A tuple *(observation, reward, done, info)*, where observation is equal to `last_observation()`; *reward (float)* is the value returned by the reward function; *done (bool)* indicates whether the episode has ended; *info (dict)* is an empty dictionary.

B.3.1 OpenAI Gym Environment

class GymEnvironment(**args: Any*, ***kwargs: Any*)
Bases: `environment.Environment`, `gym.Env`

A sub-class of Environment compliant with the OpenAI Gym API

(As described in `https://github.com/openai/gym/blob/master/docs/api.md`)
Parameters

- **connector** – The connector used to communicate with the simulator
- **actions** – A list of possible actions which will form the action space. This list will be flattened using `flatten()`, so it can contain lists or tuples of actions.
- **observed_properties** – The list of properties that will form the observation space. The order of properties will be preserved in the observation space mappings returned by `step()`, `reset()`, `last_observation()`, and `last_observation_raw()`.
- **reward** – The reward function which will be called after every step. An instance of this class will be passed as a single argument; the function should return the reward as a float. Besides serving as a reward signal, it can also recognise termination conditions and call `end_episode()`.
- **episode_initial_state** – Either a dictionary that maps property identifiers to the values these properties should take, or a callable which returns such a dictionary (without taking any arguments). This dictionary will be passed to `connectors.Connector.set_properties()` at the beginning of every new episode. For instance, it can be used in order to initialize every episode with a random state.
- **episodes_file** – A file-like object where information about every episode will be written. If None, nothing will be writen.
- **steps_file** – A file-like object where information about every step will be written. If None, nothing will be writen.
- **action_duration** – How long to wait at the end of every step. Note that if communication is slow, it may not be necessary to wait at all.

action_space
A description of the shape of the action space.

observation_space
A description of the shape of the observation space.

reward_range
The interval of the reward function. Default: (-∞, +∞)

render(*mode: Literal['console'] = 'console'*) → None
Prints the last observation as a tuple.
Raises **NotImplementedError** – if mode != 'console'.

property unwrapped: ***gym_environment.GymEnvironment***
Equals self.

B.4 Properties

B.4.1 Interface

class ObservedProperty

Bases: abc.ABC

Interface for a property used in the observation space

abstract normalize(*value: Union[None, bool, int, float, str]*) → Union[None, bool, int, float, str]

Should normalize a value for use by an RL algorithm

abstract property property_id: str

Should return the identifier of the underlying Connector property

B.4.2 Implementations

class RawObservedProperty(*property_id: str*, *space: Optional[gym.Space] = None*, *imputed_value: Union[None, bool, int, float, str] = None*)

Bases: *properties.ObservedProperty*

Provides the raw value as retrieved from the connector with no data abstraction

This implementation does not perform any discretization or normalization of the value obtained from the Connector. It is suitable for categorical properties.

Parameters

- **property_id** – The property to retrieve
- **space** – The OpenAI Gym space which describes the property. It is not needed if you don't use Gym.
- **imputed_value** – The value this property should take if the connector did not provide a value.

class DiscretizedNormalizedProperty(*property_id: str*, *min_value: Union[int, float]*, *max_value: Union[int, float]*, *n_buckets: int*, *imputed_value: Optional[Union[int, float]] = None*)

Bases: *properties.ObservedProperty*

Discretizes and normalizes a value.

This implementation will discretize a value using *discretize()* and *normalize()*.

Parameters

- **property_id** – The property to retrieve
- **min_value** – The lower end of the chosen range for this variable. Values below min_value will be discretized in the same way as min_value.
- **max_value** – The higher end of the chosen range for this variable. Values above max_value will be discretized in the same way as max_value.
- **n_buckets** – The number of buckets into which to discretize the variable.
- **imputed_value** – The value this property should take if the connector did not provide a value.

class CustomBucketProperty(*property_id: str*, *bucket_values: Sequence[Union[int, float]]*, *imputed_value: Optional[Union[int, float]] = None*)

Bases: *properties.ObservedProperty*

Discretizes a variable by placing it into the closest among a list of custom buckets.

Parameters

- **`property_id`** – The property to retrieve
- **`bucket_values`** – The values of the buckets. The result of `normalize()` will be the index of the bucket closest to the passed value.
- **`imputed_value`** – The value this property should take if the connector did not provide a value.

B.5 Recreating past episodes

class Replayer(*testing_problem:* testing_problem.TestingProblem, *episodes_file: IO*, *steps_file: IO*)

Bases: `object`

Attempts to recreate past runs

Parameters

- **testing_problem** – An instance of the TestingProblem class that was used for the original runs
- **episodes_file** (*file-like object*) – The episodes file produced by Environment
- **steps_file** (*file-like object*) – The steps file produced by Environment

replay(*episode_number: int*)

Start replaying an episode

Parameters **episode_number** – The episode to replay

B.6 Statistics on past episodes

class InfoStep(*n_episode: SupportsInt*, *time: SupportsFloat*, *action: SupportsInt*, *reward: SupportsFloat*, *observation: Tuple[Union[None, bool, int, float, str]]*)

Bases: `object`

Represents a single step of an RL episode (action, reward, observation, etc.)

n_episode
The number of the episode in which this step occurred
Type int

time
The time (in seconds since the start of the episode) at which this step occurred
Type float

action
The index of the action that was executed (at the time specified by *time*)
Type int

reward
The reward that was computed after the action was performed
Type float

observation
The raw, i.e. non-normalized state of the observed properties after the action was performed
Type Tuple[PropertyType]

class InfoEpisode(*n_episode: SupportsInt*, *end_code: SupportsInt*, *message: str*, *initial_state: Dict[str, Union[None, bool, int, float, str]]*, *steps: List[*stats.InfoStep*]*)

Bases: `object`

Represents an RL episode

n_episode
The number of the episode

end_code
An integer code representing what caused the episode to end
Type int

message
A string code representing what caused the episode to end
Type str

initial_state
Type PropertyDict

steps
The state of the observed properties after the action was performed

property duration
The duration of the episode in seconds

class Analyzer(*testing_problem:* testing_problem.TestingProblem, *episodes_file: IO*, *steps_file: IO*)

Bases: `object`

Analyzes data from past runs.
Parameters
- **testing_problem** – An instance of the TestingProblem class that was used for the original runs
- **episodes_file** (*file-like object*) – The episodes file produced by Environment

- **steps_file** (*file-like object*) – The steps file produced by Environment

count_actions() → collections.Counter
Count how many times each action was executed.

average_rewards() → Dict[*actions.Action*, float]
Calculates the average reward for each action

episode_lengths(*end_codes: Container = Interval(- inf, inf, closed='right')*) → List[int]
Returns a list containing the number of steps in every matching episode
Parameters **end_codes** – end codes to consider

failure_rate(*failing_codes: Container = Interval(- inf, - 1, closed='right')*) → float
Calculates the proportion of episodes with codes in end_codes
Parameters **failing_codes** – Codes to consider. Can be a list of codes, a range, a Numpy Interval, ...

Returns 1.0 if all codes are in failing_codes NaN if episodes_file was empty

episode_adversity(*episode:* stats.InfoEpisode, *integrand: Literal['x', 'x^2', 'abs(x)'] = 'x'*, *start_step: Optional[int] = 0*, *stop_step: Optional[int] = None*, *start_property: Optional[int] = 0*, *stop_property: Optional[int] = None*) → pandas.core.frame.DataFrame
Calculates the adversity faced in an episode with respect to all observed properties

According to the value of the integrand parameter, the adversity faced in an episode with respect to observed property x can be the integral of x, x^2, or abs(x) with respect to time. Since the observations are time-discrete, the integral is approximated by the trapezoidal rule[1]. Note that adversity is only defined for episodes with two or more steps.

If the values of an observed property can be negative, they could cancel each other in the calculation of the integral. If this is not appropriate, an even integrand function like x^2 or abs(x) can be used instead. Using x^2 amplifies the impact of outlying data points; if this is undesirable, abs(x) can be used instead.

Adversity is only defined for numeric properties. If the observation space also contains non-numeric properties, take care to exclude them. None or NaN values in the included properties will also cause errors; if they are present, they need to be removed first by calling :meth:~`impute_missing_values`.

By default, adversity is calculated for all the steps of an episode. A subinterval of steps can also be specified using start_step and stop_step. These parameters have the same semantics as in Python slices. Likewise, if you only wish to calculate adversity for a subset of observed properties, you can specify *start_property* and *property_stop*, in which case only properties in *testing_problem.observed_properties[start_property:property_stop]* will be calculated.

[1] See https://en.wikipedia.org/wiki/Trapezoidal_rule
Parameters
- **episode** – The episode for which adversity is computed.
- **start_step** – Indicates the start of the subinterval for which adversity is computed.
- **stop_step** – Indicates the end of the subinterval for which adversity is computed.
- **integrand** – The function to be integrated. Possible values: 'x','x^2','abs(x)', where x is the value of a property at any given step.
- **start_property** – Indicates the start of the slice from *testing_problem.observed_properties* to consider.

- **stop_property** – Indicates the start of the slice from *testing_problem.observed_properties* to consider.

Returns A DataFrame with one row, whose label is the episode number, and a column for each observed property of the testing problem, whose label is the id of that property. Every cell of the dataframe contains the adversity with respect to the indicated property.

Raises
- **ValueError** – if the episode (or the indicated subinterval) contains less than two steps.
- **ValueError** – if integrand is not among the possible values.

adversity(*end_codes: Container = Interval(- inf, inf, closed='right')*, *integrand: Literal['x', 'x^2', 'abs(x)'] = 'x'*, *start_step: Optional[int] = 0*, *stop_step: Optional[int] = None*, *start_property: Optional[int] = 0*, *stop_property: Optional[int] = None*) → pandas.core.frame.DataFrame

Calculates the adversity over all episodes with matching end codes.

This is a convenience function which codes `episode_adversity()` for every matching episode with more than two steps and aggregates the result in a single DataFrame. The parameters, the format of the returned DataFrame, and the exceptions raised are all as in the documentation of `episode_adversity()`.

Parameters **end_codes** – Codes to consider. Can be a list of codes, a range, a Numpy Interval, ...

adversity_stats(*end_codes: Container = Interval(- inf, - 1, closed='right')*, *integrand: Literal['x', 'x^2', 'abs(x)'] = 'x'*, *start_step: Optional[int] = 0*, *stop_step: Optional[int] = None*, *start_property: Optional[int] = 0*, *stop_property: Optional[int] = None*) → pandas.core.frame.DataFrame

Calculates aggregate statistics about adversity over matching episodes.

In the background, this method calls `episode_adversity()` for every matching episode with more than two steps and aggregates the result in a single DataFrame. The parameters and the exceptions raised are all as in the documentation of `episode_adversity()`.

This function attempts to help calculate the amount of "adverse conditions" encountered during the episode. For each failing episode (i.e., for each episode with an end code inside end_codes), and for each observed property, it sums all the observations in the episode. Then, for each observed property, it calculates the minimum, maximum, mean, and standard deviation of the sums across all failing episodes. If there are no failing episodes, an empty DataFrame() is returned.

Parameters **end_codes** – Codes to consider. Can be a list of codes, a range, a Numpy Interval, ...

Returns A DataFrame with a row for every observed property and with the following columns: 'min', 'max', 'mean', 'stdev'. If no matching episodes with at least two steps were found, an empty DataFrame.

impute_missing_values(*imputed_values: Optional[Tuple[Union[None, bool, int, float, str], ...]] = None*) → None

Imputes missing (i.e. None and NaN) values.

This method replaces None and NaN observations, modifying them in-place. It does this as follows:

1. The value observed (or already imputed) for the previous step is used, if there is one.

2. If a None or NaN value was observed for a property in the first step of an episode, then the corresponding value provided in the *imputed_values* parameter is used. This value will be used even if it is None or NaN.
3. If imputed_values is not provided, Property.imputed_value is used. This value will be used even if it is None or NaN.

   Parameters **imputed_values** – A list of values to impute. They must be in the same order as *testing_problem.observed_properties*.

write(*episode_file: IO*, *steps_file: IO*) → None

Write the modified data to files.

Parameters

- **episodes_file** (*file-like object*) – The file in which episodes will be logged
- **steps_file** (*file-like object*) – The file in which steps will be logged

B.7 RL model for tabular algorithms

class RLAlgorithm(*value*)

Bases: enum.Enum

A tabular algorithm for use in TabularModel

QLearning

The Q-Learning algorithm (Watkins, 1989)

SARSA

The State–action–reward–state–action algorithm (Rummery and Niranjan, 1994)

class TabularModel(*environment: Union[*environment.Environment*, gym.Env]*, *algorithm:* tabular_model.RLAlgorithm = *RLAlgorithm.SARSA*, *learning_rate: float = 0.6*, *discount_factor: float = 0.9*, *epsilon: float = 0.1*, *backup_frequency: Optional[int] = None*, *sort_observation_elements: bool = False*)

Bases: object

A model which implements Q-Learning and Sarsa on any gym-like environment.

All the attributes of this class can also be modified during runtime, except for the algorithm. For example, you can gradually decrease the learning rate.

Parameters

- **environment** – The environment on which to operate
- **algorithm** – The algorithm to use
- **learning_rate** – Determines how quickly newly learned information replaces old information
- **discount_factor** – Determines how much future rewards are valued compared to immediate ones
- **epsilon** – Determines how often random actions are explored rather than exploiting
- **backup_frequency** – How often the model should backup (by calling `backup()`). For example, *backup_frequency=3* indicates it should be backed up after every three episodes. If this parameter is not provided, `backup()` is never called automatically.
- **sort_observation_elements** – If this is true, the properties in the observation space returned by the environment will be sorted alphabetically during the conversion to a tuple. If *environment* is an instance of `environment.Environment` or `gym_environment.GymEnvironment`, this is not necessary, since these classes always provide a consistently ordered OrderedDict. If you use a custom environment, you can set this parameter to True. For the learning, the order of properties in the observation dictionary does not need to be alphabetic, it just needs to be consistent.

environment

See above.

algorithm

See above.

learning_rate

See above.

discount_factor

See above.

epsilon

See above.

current_episode
The number of the current episode.

max_episodes
The maximum number of episodes, after which training or testing will stop.

table
The underlying *QLearningTable* / *SarsaTable*.
Type typing.Union[QLearningTable, SarsaTable]

static load (*filename*, **args*, ***kwargs*) → *tabular_model.TabularModel*
Loads and returns a new model from a file. After filename, are arguments are passed to TabularModel.__init__.
Parameters
- **filename** – The name of the file to read
- **args** – Positional arguments as in TabularModel.__init__
- **kwargs** – Keyword arguments as in TabularModel.__init__

Returns A new TabularModel with the given parameters.

load_in_place (*filename='Q.pickle'*) → None
Loads model from a file in-place.
Parameters **filename** – The name of the file to read

save (*filename: str = 'Q.pickle'*) → None
Saves model to a file
Parameters **filename** – The name of the file to save to

backup () → None
Saves model with an automatic filename containing the timestamp.

learn (*max_episodes: int = - 1*)
Train the model for a pre-defined number of episodes.
Parameters **max_episodes** – The number of episodes to train for. If set to -1, training will continue until interrupted. To do so, set *model.max_episodes = model.current_episode*.

test (*testing_episodes: int = - 1*, *testing_epsilon: float = 0.0*, *restart_numbering: bool = False*) → None
Use model for testing for a pre-defined number of episodes.

The difference between training and testing is that the model will not be updated, i.e. no learning will take place during the execution of this method.
Parameters
- **testing_episodes** – The number of episodes to train for. If set to -1, training will continue until interrupted. To do so, set *model.max_episodes = model.current_episode*.
- **testing_epsilon** – The epsilon value to use during testing. If 0.0, the model will be greedily exploited. Before this method returns, model.epsilon will be reset to its previous value. If this parameter is set to None, model.epsilon will be used instead, without changing it at all.
- **restart_numbering** – If True, the numbering of the episodes will restart.

B.8 Testing problem

class TestingProblem

Bases: abc.ABC

An abstract class representing a TEA testing problem.

Definitions of RL testing configurations should be defined as subclasses of this class. This class is used by tea-analyze and tea-replay in order to extract the required information about a testing problem. Note that setting the attributes of this subclass is not a substitute for passing them to the environment, connector, etc.

actions
The sequence of actions that may be taken by the RL algorithm

observed_properties
The properties that will form the observation space

episode_initial_state
The initial state of each episode, or a callable which generates it.

connector
The simulator connector.

reward
The reward function.

environment
The environment which will represent the system under test

model
The RL model.

abstract run() → None
This method is required; it should contain any code necessary to execute the testing problem.

abstract analyze(*episodes_file: IO*, *steps_file: IO*) → None
This method is optional; if provided, it will be called by tea-analyze instead of producing the default stats.

B.9 Utility classes and functions

approximate (*x: SupportsFloat*, *allowedValues: Sequence[SupportsFloat]*) → SupportsFloat

Approximates x to the closest value in allowedValues

Parameters

- **x** – the number to approximate
- **allowedValues** – a sequence of allowed values.

Returns an element in allowedValues

normalize (*x: SupportsFloat*, *zero: SupportsFloat*, *one: SupportsFloat*, *allow_out_of_bounds: bool = False*) → float

Normalizes a float or an int on a scale from *zero* to *one*.

Parameters

- **x** – number to normalize
- **zero** – lower end of the scale, normalized as 0.0
- **one** – higher end of the scale, normalized as 1.0
- **allow_out_of_bounds** – determines whether values outside [zero, one] are allowed; if False, either 0.0 or 1.0 is returned, whichever is closer.

Returns the normalized float

Examples:

```
normalize(60, zero=0, one=100) -> 0.6
normalize(120, zero=0, one=100) -> 1.0
normalize(120, zero=0, one=100, allow_out_of_bounds=True) -> 1.2
normalize(6, zero=5, one=10) -> 0.2
normalize(4, zero=5, one=10, allow_out_of_bounds=True) -> -0.2
```

discretize (*x: SupportsFloat*, *buckets: int*) → int

Discretizes an element of the continuous interval [0, 1]. Returns an integer element of the discrete interval [0, buckets).

In practice, this function cuts the interval [0, 1] into subintervals, the number of which is the parameter "buckets", and assigns each interval an integer n: 0 <= n < buckets.

Parameters

- **x** – The continuous number to discretize
- **buckets** – The number of *buckets* (i.e., subintervals)

Examples:

```
discretize(0.1618033988749895, buckets=5) -> 0
discretize(0.5, buckets=5) -> 2
discretize(1, buckets=5) -> 4
```

static_var (***kwargs*)

Function decorator used to create 'static variables' inside a function (that is, variables that retain their values between function call).

Example:

```
@static_var(counter=0)
def foo():
    foo.counter += 1
    print('This function has been called %d times' % foo.counter)
```

Note that the foo.counter attribute can also be accessed from outside the function. The function object will also have the following attributes:

`default_static_vars`: a dict mapping all the variables to their default (original) values
`reset_static_vars()`: a method that resets all the variables to their default (original) values

Note that if these attributes refer to the original *objects* that were passed to the decorator. I.e., if you passed a mutable object (e.g. a list), but did not reassign the reference to another object, then the value contained in *default_static_vars* will be the current value, and *reset_static_vars* will not have an effect on it. For more information, see: `https://docs.quantifiedcode.com/python-anti-patterns/correctness/mutable_default_value_as_argument.html`

flatten (*it: Iterable*, *max_recursion: int = - 1*) → List

Flatten a nested iterable into a flat list

Parameters

- **it** – iterable to flatten
- **max_recursion** – the maximum level of recursion. If negative, infinite recursion is allowed.

class NormalizedScalar

Bases: `SupportsFloat`, `SupportsInt`

This class represents a quantity that is normalised to the interval [0, 1].

It cannot be instantiated directly; create a subclass using the class method NormalizedScalar.create_subclass instead. This class implements comparison, addition, subtraction, multiplication and division with type checking. I.e., two instances of the same subclass of NormalizedScalar can be compared, added, or subtracted, and an instance of a subclass can be multiplied or divided by an int. For all other operations, an explicit conversion to float is required.

Example usage:

```
Percentage = NormalizedScalar.create_subclass('Percentage', 0.0,␣
↪100.0, False)
p = Percentage.normalize(30)
p + p * 2 == Percentage(0.90) -> True
```

denormalize() → float

Returns a denormalized number, i.e. one brought back to the original scale of the quantity.

static create_subclass (*name: str*, *zero: SupportsFloat*, *one: SupportsFloat*, *allow_out_of_bounds: bool = False*) → type

Creates a subclass of NormalizedScalar.

Parameters

- **name** – The __name__ of the new subclass
- **zero** – the lower end of the scale
- **one** – the higher end of the scale
- **allow_out_of_bounds** – determines whether values outside [zero, one] are allowed

These parameters have the same semantics as for *normalize()*; see examples there.

Returns (type) A subclass of NormalizedScalar with the provided parameters

C FlightGear properties

An exploratory study was conducted prior to the problem definition presented in section 4.1.1 with the goal of identifying the FlightGear properties most suitable to perform testing based on reinforcement learning and FlightGear. The main result of the exploratory study is a list of properties that we report in Section C.1 along with suggested discretization strategies for the observation space and encoding strategies for the action space. The list is a superset of the properties presented in section 4.1.1, they might be used to test requirements of the IT-AUTOFLIGHT component different than the one considered in section 4.1.1; also, they might be used to test other software components that can be executed through FlightGear (e.g., the PX4 Autopilot controller[10]). In Section C.2, instead, we report the list of all the FlightGear properties that were tested.

C.1 Selected properties

Property	Comments
/environment/clouds/layer[N]/coverage	Possible values: “clear”, “few”, “scattered”, “broken”, “overcast”, “cirrus”. Bear in mind the existence of different layers of clouds. Suggested discretization for observation space: CLEAR, FEW, CIRRUS, SCATTERED, BROKEN, OVERCAST. Suggested encoding for action space: INCREASE, DECREASE.
/environment/config/presets/turbulence-magnitude-norm	Range: 0-1.0 This property is only effective if /environment/config/presets/turbulence-magnitude-norm-override has been set to true. Suggested discretization for observation space: NONE = 0, LIGHT = 0.33, MEDIUM = 0.66, SEVERE = 1.00. Suggested encoding for action space: INCREASE, DECREASE.
/environment/config/presets/visibility-m	Effective only if the /environment/config/presets/visibility-m-override property is set. The change is immediately visually perceivable by the user; however, IT-AUTOFLIGHT does not seem to take it into account in any obvious way. Suggested discretization for observation space: NONE = 0, POOR = 100, LIMITED = 1000, FAR = 10000. Suggested encoding for action space: INCREASE, DECREASE.

[10] See `https://docs.px4.io/master/en/simulation/flightgear.html`

/environment/weather-scenario	Possible values: • “Live data”, • “Manual input”, • “Core high pressure region”, • “High pressure region”, • “Border of a high pressure region”, • “Border of a low pressure region”, • “Low pressure region”, • “Core low pressure region”, • “Warm sector”, • “Fair weather”, • “Thunderstorm”, • “Stormy Monday”, • “Marginal VFR”, • “CAT I minimum”, • “CAT II minimum”, • “CAT IIIb minimum”, • “Early morning fog”. Suggested encoding for observation space: subset of available options. Suggested encoding for action space: CHANGE.

/position/altitude-ft	If test objective is "reach desired altitude". Suggested discretization for observation space: $ALT_0 : \Delta < 100m$; $ALT_N :\in [100 + N < \Delta < 200 + N]$. $\Delta = (GOAL - current_altitude)$ Suggested encoding for action space: Should not be included in action space Suggested reward: Change in vertical speed since last action (actions are performed approximately every 2 seconds) Failure detection: Case 1: if the average vertical speed in the last 20 seconds is lower than the average vertical speed according to specifications, deliver maximum reward, signal the failure, but keep testing. Case 2: if the altitude drops dramatically (i.e., the airplane is in free fall), deliver maximum reward and end tge episode. If test objective is "keep desired altitude" . Suggested discretization for observation space: $ALT_0 : \Delta < 600m; ALT_N :\in [600 + N < \Delta < 2600 + N]$. Suggested encoding for action space: Should not be included in action space Suggested reward: Average distance from goal altitude computed with 5 samples taken since last action (actions are performed every 2 seconds) Failure detection: CASE1) current altitude above/below target altitude for a delta that is greater than specifications (max reward, signal the failure but keep testing) CASE2) altitude drops dramatically (i.e., the airplane is dropping to land) (max reward and episode ends).

/fdm/jsbsim/atmosphere/gust-down-fps	Sufficiently high values cause the aircraft to lose altitude and crash. Suggested discretization for observation space: approximation to an element of $-k, -0.66k, -0.33k, 0, 0.33k, 0.66k, k$ (where k is a constant reference value). Suggested encoding for action space: INCREASE, DECREASE.
/fdm/jsbsim/atmosphere/gust-east-fps	Sufficiently high values cause the aircraft to spin uncontrollably and crash swiftly. Suggested discretization for observation space: approximation to an element of $-k, -0.66k, -0.33k, 0, 0.33k, 0.66k, k$ (where k is a constant reference value). Suggested encoding for action space: INCREASE, DECREASE.
/fdm/jsbsim/atmosphere/gust-north-fps	 Changes the aircraft's heading and has an impact on the autopilot's heading hold functionality. Suggested discretization for observation space: approximation to an element of $-k, -0.66k, -0.33k, 0, 0.33k, 0.66k, k$ (where k is a constant reference value). Suggested encoding for action space: INCREASE, DECREASE.

C.2 All tested properties

All the properties below were tested (i.e., we manually verified the effect on a flying drone and environment when their values were modified) using FlightGear 2020.3.9 and IT-AUTOFLIGHT V 4.0.X. For each of them, we report if an effect was observed or not.

Property	Comments
/ai/models/aircraft/controls/flight/lateral-mode	No effect was observed
/ai/models/aircraft/controls/flight/lateral-mode	No effect was observed
/ai/models/aircraft/controls/flight/longitude-mode	No effect was observed
/ai/models/aircraft/controls/flight/target-alt	No effect was observed
/ai/models/aircraft/controls/flight/target-hdg	No effect was observed
/ai/models/aircraft/controls/flight/target-pitch	No effect was observed
/ai/models/aircraft/controls/flight/target-roll	No effect was observed
/ai/models/aircraft/controls/flight/target-spd	No effect was observed
/ai/models/aircraft/controls/flight/vertical-mode	No effect was observed
/consumables/fuel/tank[N]/level-gal_imp /consumables/fuel/tank[N]/level-gal_us /consumables/fuel/tank[N]/level-kg /consumables/fuel/tank[N]/level-lbs /consumables/fuel/tank[N]/level-m3	Effective. These can be modified in flight to top up the fuel; however, if they are modified after fuel has run out and the engines have stopped, this will not restart the engines.
/consumables/fuel/total-fuel-kg	Read-only
/controls/flight/aileron	Continuously set by IT-AUTOFLIGHT; in practice read-only when the latter is enabled.
/controls/flight/aileron-trim	Effective. IT-AUTOFLIGHT does not control this, which is a bug in my opinion (especially because it does control elevator-trim). It does quickly change aileron, but this is not always enough to stabilise the aircraft.
/controls/flight/elevator	Continuously set by IT-AUTOFLIGHT; in practice read-only when the latter is enabled.
/controls/flight/elevator-trim	Continuously changed by autopilot, however, if quickly set many times, autopilot will maintain it to a similar level. Crashes the airplane. Range: -1.0 to 1.0.
/controls/flight/rudder	Continuously set by IT-AUTOFLIGHT; in practice read-only when the latter is enabled.
/controls/flight/rudder-trim	Effective, even on autopilot. Range: -1 to 1. Seems to make it a bit more difficult for the autopilot to climb/maintain altitude.
/controls/flight/wing-fold	No effect was observed
/controls/flight/wing-sweep	No effect was observed
/controls/gear/steering	No effect was observed.
/controls/gear/tailhook	No effect was observed (probably because the MQ-9 does not have a tail hook)
/environment/aircraft-effects/fog-level	No effect was observed
/environment/aircraft-effects/frost-level	No effect was observed

/environment/clouds/layer[N]//coverage	Effective. Possible values: “clear”, “few”, “scattered”, “broken”, “overcast”, “cirrus” (check). Bear in mind the existence of different layers of clouds.
/environment/clouds/layer[N]//coverage-type	0 (“overcast”), 1 (“broken”), 2 (“scattered”), 3 (“few”), 4 (“cirrus”), 5 (“clear”). Bear in mind the existence of different layers of clouds.
/environment/clouds/layer[N]//elevation-ft	Effective
/environment/clouds/layer[N]//layer-type	No effect was observed
/environment/clouds/layer[N]//thickness-ft	Cloud thickness. Range: 0-1000.
/environment/clouds/layer[N]//transition-ft	No effect was observed
/environment/clouds/layer[N]//visibility-m	No effect was observed
/environment/clouds/layer/alpha	No effect was observed. Range: 0 to 1.0
/environment/config/aloft/entry[N]/ turbulence/magnitude-norm	Effective, as long as /environment/config/presets/turbulence-magnitude-norm-override is false. Range : 0-1.0. However, it sometimes loses its effect after a while; the reason for this is unknown. Use of /environment/config/presets/turbulence-magnitude-norm is recommended instead.
/environment/config/presets/turbulence-magnitude-norm	Effective. Range: 0-1.0 Suggested discretization: NONE = 0, LIGHT = 0.33, MEDIUM = 0.66, SEVERE = 1.00 This property is only effective if /environment/config/presets/turbulence-magnitude-norm-override has been set to true (either manually or via the command line)
/environment/config/presets/turbulence-magnitude-norm-override	(Boolean) Effective. Controls whether /environment/config/presets/turbulence-magnitude-norm has effect. When enabling this property, take care to set it to the correct data type (i.e., using setb in the Telnet interface; this is not possible using the FlightGear internal property browser). If the property is set as a string, any consequent attempt to change it, even with the correct data type, fails.
/environment/config/presets/visibility-m	Effective, if the /environment/config/presets/visibility-m-override property is set. Effective in the sense that the change is immediately visually perceivable by the user; however, IT-AUTOFLIGHT does not seem to take it into account in any obvious way.
/environment/config/presets/visibility-m-override	Controls whether /environment/config/presets/visibility-m is taken into account.
/environment/contrails	No effect was observed
/environment/effective-visibility-m	No effect was observed
/environment/fog-structure	No effect was observed
/environment/metar/base-wind-speed-kt	No effect was observed, even when live updating is off
/environment/metar/dewpoint-degc	No effect was observed
/environment/metar/dewpoint-sea-level-degc	No effect was observed
/environment/metar/gust-wind-speed-kt	No effect was observed, even when live updating is off
/environment/metar/hail-norm	No effect was observed
/environment/metar/max-visibility-m	No effect was observed
/environment/metar/min-visibility-m	No effect was observed

/environment/metar/pressure-inhg	No effect was observed
/environment/metar/pressure-sea-level-inhg	No effect was observed
/environment/metar/rain-norm	No effect was observed
/environment/metar/rel-humidity-norm	No effect was observed
/environment/metar/set-ground-cloud-layer	No effect was observed
/environment/metar/snow-cover	No effect was observed
/environment/metar/snow-norm	No effect was observed
/environment/metar/temperature-degc	No effect was observed
/environment/metar/temperature-sea-level-degc	No effect was observed
/environment/metar/time-to-live	Countdown to the next update. Runs even when live updating is off.
/environment/metar/valid	No effect was observed. Does not become false when, for example, physically impossible values are set to the other properties in this directory. It seems to become false when /environment/metar/time-to-live runs out (even if live updating is off). It is not known whether this has an effect when live updating is off.
/environment/metar/visibility/max-m	No effect was observed.
/environment/metar/visibility/min-m	No effect was observed.
/environment/params/external-precipitation-level-m	No effect was observed
/environment/params/external-precipitation-level-m	No effect was observed, even if /environment/params/use-external-precipitation-level is set to true
/environment/params/fog-mist-haze-layer/fog-shallow-thickness-ft	No effect was observed
/environment/params/fog-mist-haze-layer/fog-thickness-ft	No effect was observed
/environment/params/metar-updates-environment	Controls live weather updating. Disable when controlling the corresponding weather aspect manually.
/environment/params/metar-updates-snow-level	Controls live weather updating. Disable when controlling the corresponding weather aspect manually.
/environment/params/metar-updates-winds-aloft	Controls live weather updating. Disable when controlling the corresponding weather aspect manually.
/environment/params/precipitation-level-ft	Read-only
/environment/params/precipitation-level-ft	Read-only
/environment/params/use-external-precipitation-level	Unknown effect
/environment/params/use-external-precipitation-level	(Boolean) No effect was observed
/environment/turbulence/magnitude-norm	Read-only; can be set indirectly, e.g. by setting /environment/config/presets/turbulence-magnitude-norm. Equal to rs, where r = /environment/turbulence/raw-magnitude-norm and s = /environment/turbulence/sensitivity. By setting s to a negative value, arbitrarily high values can be achieved for this property, but this does not appear to have an effect on the system. The effective range of this property is therefore 0.0-1.0.

/environment/turbulence/raw-magnitude-norm	Read-only.
/environment/turbulence/raw-magnitude-norm	Read-only; can be set indirectly, e.g. by setting /environment/config/presets/turbulence-magnitude-norm. Range: 0-1.0.
/environment/turbulence/sensitivity	The exponent to which raw-magnitude-norm is raised when calculating magnitude-norm. See /environment/turbulence/magnitude-norm
/environment/weather-scenario	Effective. Possible values: • “Live data”, • “Manual input”, • “Core high pressure region”, • “High pressure region”, • “Border of a high pressure region”, • “Border of a low pressure region”, • “Low pressure region”, • “Core low pressure region”, • “Warm sector”, • “Fair weather”, • “Thunderstorm”, • “Stormy Monday”, • “Marginal VFR”, • “CAT I minimum”, • “CAT II minimum”, • “CAT IIIb minimum”, • “Early morning fog”
/environment/wildfire /enabled	No effect was observed
/environment/wildfire/models/enabled	No effect was observed
/environment/wind-from-down-fps	Read-only
/environment/wind-from-east-fps	Read-only
/environment/wind-from-heading-deg	Read-only
/environment/wind-from-north-fps	Read-only
/environment/wind-speed-kt	Read-only; controllable indirectly, e.g. through command line argument
/fdm/jsbsim/atmosphere/gust-down-fps	Sufficiently high values cause the aircraft to lose altitude and crash.
/fdm/jsbsim/atmosphere/gust-east-fps	Sufficiently high values cause the aircraft to spin uncontrollably and crash swiftly.
/fdm/jsbsim/atmosphere/gust-north-fps	Changes the aircraft’s heading and has an impact on the autopilot’s heading hold functionality.
/fdm/jsbsim/atmosphere/total-wind-down-fps	Read-only
/fdm/jsbsim/atmosphere/total-wind-east-fps	Read-only
/fdm/jsbsim/atmosphere/total-wind-north-fps	Read-only
/fdm/jsbsim/atmosphere/turb-down-fps	Read-only
/fdm/jsbsim/atmosphere/turb-east-fps	Read-only
/fdm/jsbsim/atmosphere/turb-gain	No effect was observed
/fdm/jsbsim/atmosphere/turb-gain	No effect was observed
/fdm/jsbsim/atmosphere/turb-north-fps	Read-only
/fdm/jsbsim/atmosphere/turb-rate	No effect was observed

/fdm/jsbsim/atmosphere/turb-rhythmicity	No effect was observed
/fdm/jsbsim/atmosphere/turbulence/milspec/severity	Read-only
/fdm/jsbsim/atmosphere/turbulence/milspec/windspeed_at_20ft_AGL-fps	Read-only
/fdm/jsbsim/atmosphere/vapor-fraction-ppm	Read-only
/fdm/jsbsim/atmosphere/vapor-fraction-ppm	Read-only
/fdm/jsbsim/atmosphere/wind-down-fps	Read-only
/fdm/jsbsim/atmosphere/wind-east-fps	Read-only
/fdm/jsbsim/atmosphere/wind-mag-fps	Read-only
/fdm/jsbsim/atmosphere/wind-north-fps	Read-only
/orientation/heading-deg	Effective. Sufficiently brusque changes (e.g. 180 degrees) cause a crash.
/orientation/pitch-deg	Effective. Sufficiently brusque changes (e.g. 180 degrees) cause a temporary loss of altitude.
/orientation/roll-deg	Effective. Sufficiently brusque changes (e.g. 180 degrees) cause a crash.
/orientation/yaw-deg	No effect was observed
/position/altitude-ft	Effective.
/sim/rendering/precipitation-enable	No effect was observed

References

[1] IEEE standard glossary of modeling and simulation terminology. *IEEE Std 610.3-1989*, pages 1–0, 1989.

[2] Mohamed Abdel-Basset, Laila Abdel-Fatah, and Arun Kumar Sangaiah. Chapter 10 - metaheuristic algorithms: A comprehensive review. In Arun Kumar Sangaiah, Michael Sheng, and Zhiyong Zhang, editors, *Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications*, Intelligent Data-Centric Systems, pages 185–231. Academic Press, 2018.

[3] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74, 2016.

[4] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. *Proceedings - International Conference on Software Engineering*, pages 1016–1026, 2018.

[5] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 2–8, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Oliver Möller Adrian Pop, Florian Lapschies. INTO-CPS D5.2a: Test automation module in the INTO-CPS Platform. `https://into-cps.org/publications/`, 2015.

[7] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, Andi Domi, and Ivan Porres. Using deep reinforcement learning for exploratory performance testing of software systems with multi-dimensional input spaces. *IEEE Access*, 8:195000–195020, 2020.

[8] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9447:194–207, 2015.

[9] Guilherme Aschauer, Alexander Schirrer, and Martin Kozek. Co-simulation of Matlab and FlightGear for identification and control of aircraft. *IFAC-PapersOnLine*, 28(1):67–72, 2015.

[10] BeamNG GmbH. Beamng game simulator. https://www.beamng.com, 2021.

[11] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 63–74, 2016.

[12] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154, 2018.

[13] Christopher Brooks, Fabio Cremona, Edward A. Lee, David Lorenzetti, Thierry Nouidui, and Michael Wetter. Cyphysim - a cyber-physical systems simulator, October 2015. Presented at the Eleventh Biennial Ptolemy Miniconference.

[14] CARLA community. CARLA simulator. https://carla.org/, 2021.

[15] Yuqi Chen, Bohan Xuan, Christopher M. Poskitt, Jun Sun, and Fan Zhang. Active fuzzing for testing and securing cyber-physical systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 14–26, New York, NY, USA, 2020. Association for Computing Machinery.

[16] COSIMATE technologies. Cosimate simulator. https://www.cosimate.com/, 2021.

[17] Indraneel Das and J. E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM J. on Optimization*, 8(3):631–657, March 1998.

[18] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[19] FlightGear community. FlightGear open source simulator. https://www.flightgear.org/, 2021.

[20] FlightGear community. IT-AUTOFLIGHT autonomous flight system. https://wiki.flightgear.org/IT-AUTOFLIGHT, 2021.

[21] Alessio Gambi, Marc Mueller, and Gordon Fraser. AsFault: Testing self-driving car software using search-based procedural content generation. *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*, pages 27–30, 2019.

[22] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283, 2019.

[23] Juan Francisco Garcia, Daniel Jurjo, Fernando Mac, and Alessandra Gorla. An application of KLEE to aerospace industrial software. pages 1–7.

[24] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A. Wright, Joseph E. Gonzalez, and Ion Stoica. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles, 2021.

[25] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: A survey. *ACM Computing Surveys*, 51(3), 2018.

[26] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. Enabling model testing of cyber-physical systems. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, page 176–186, New York, NY, USA, 2018. Association for Computing Machinery.

[27] F. U. Haq, D. Shin, S. Nejati, and L. C. Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 85–95, 2020.

[28] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel Briand. Can offline testing of deep neural networks replace their online testing? *Empirical Software Engineering*, 90(26).

[29] Siddhartha Kumar Khaitan and James D. McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, 2015.

[30] Junhwi Kim, Minhyuk Kwon, and Shin Yoo. Generating test input with deep reinforcement learning. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 51–58, 2018.

[31] Moritz Klischat, Edmond Irani Liu, Fabian Holtke, and Matthias Althoff. Scenario factory: Creating safety-critical traffic scenarios for automated vehicles. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–7, 2020.

[32] Mark Koren, Saud Alsaif, Ritchie Lee, and Mykel J. Kochenderfer. Adaptive stress testing for autonomous vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1–7, 2018.

[33] Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, and Andrey Sadovykh. Integrated tool chain for model-based design of cyber-physical systems: The into-cps project. In *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*, pages 1–6, 2016.

[34] Tao Ma, Shaukat Ali, Tao Yue, and Maged Elaasar. Testing self-healing cyber-physical systems under uncertainty: A fragility-oriented approach. *Software Quality Journal*, 27(2):615–649, jun 2019.

[35] Mathworks. Simulink simulator. https://www.mathworks.com/products/simulink.html, 2021.

[36] Vuong Nguyen, Stefan Huber, and Alessio Gambi. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 128–135, 2021.

[37] Opal-rt technologies. Opal-rt. real-time simulator for power systems, automotive, power electronics, aerospace. https://www.opal-rt.com/, 2021.

[38] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 153–164, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.

[40] Maxime Perrotin, Eric Conquet, Julien Delange, André Schiele, and Thanassis Tsiodras. Taste: A real-time software engineering tool-chain overview, status, and future. In Iulian Ober and Ileana Ober, editors, *SDL 2011: Integrating System and Software Modeling*, pages 26–37, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (2):876–888, 2020.

[42] Hendrik Roehm, Jens Oehlerking, Matthias Woehrle, and Matthias Althoff. Model conformance for cyber-physical systems: A survey. *ACM Trans. Cyber-Phys. Syst.*, 3(3), August 2019.

[43] RTDS technologies. Rtds simulator. simulator for power systems. https://www.rtds.com/, 2021.

[44] Conor Ryan. Evolutionary algorithms and metaheuristics. In Robert A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 673–685. Academic Press, New York, third edition edition, 2003.

[45] Zahra Sadri-Moshkenani, Justin Bradley, and Gregg Rothermel. Survey on test case generation, selection and prioritization for cyber-physical systems. *Software Testing, Verification and Reliability*, n/a(n/a):e1794.

[46] Fute Shang, Buhong Wang, Tengyao Li, Jiwei Tian, and Kunrui Cao. Cpfuzz: Combining fuzzing and falsification of cyber-physical systems. *IEEE Access*, 8:166951–166962, 2020.

[47] Ekhlas Sonu and Prashant Doshi. GaTAC: A Scalable and Realistic Testbed for Multiagent Decision Making (Demonstration). *Aamas*, (June 2012):1507–1508, 2012.

[48] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[49] SysGo technologies. PikeOS. https://www.sysgo.com/pikeos, 2021.

[50] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. Openmeta: A model- and component-based design tool chain for cyber-physical systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing*, pages 235–248, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[51] Ana Turlea. Search based model in the loop testing for cyber physical systems. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 22–28, 2018.

[52] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[53] Yoriyuki Yamagata, Shuang Liu, Takumi Akazaki, Yihai Duan, and Jianye Hao. Falsification of cyber-physical systems using deep reinforcement learning. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[54] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 423–435, 2021.

[55] Ziyuan Zhong, Yun Tang, Yuan Zhou, Vania de Oliveira Neves, Yang Liu, and Baishakhi Ray. A survey on scenario-based testing for automated driving systems in high-fidelity simulation, 2021.

[56] Xin Zhou, Xiaodong Gou, Tingting Huang, and Shunkun Yang. Review on testing of cyber physical systems: Methods and testbeds. *IEEE Access*, 6:52179–52194, 2018.