Scenario-based Automatic Testing of a Machine Learning Solution with the Human in the Loop

Author information scrubbed for double-blind reviewing

Affiliation scrubbed for double-blind reviewing

Abstract. More and more applications rely on machine learning, particularly interactive online learning, to make decisions tailored to human needs and situation. Like any program, the behavior of learning programs must be verified and validated. Testing is one way to achieve this. In this paper, we analyze the challenges of testing machine learning solutions, focusing on programs that learn online and in interaction with human users. Given the issues arising from the presence of the human in the loop, the non-determinism and the dynamics of online learning, we propose a scenario-based approach. We apply it to the test of OCE, a learning program that builds applications in ambient environments with the user in the loop. In addition, two prototype tools are presented to implement test scenarios and automate their execution for the assessment OCE.

Keywords: Interactive machine learning · Human-in-the-loop machine learning and decision · Assessment · Test · Scenario

1 Introduction

As part of a research project on Ambient Intelligence [6] called "OppoCompo", a solution based on interactive machine learning (IML) has been proposed. This solution dynamically builds applications for a human user in ambient environments (Internet of Things, Smart City, etc.). The "Opportunistic Composition Engine" called OCE learns the needs and preferences of the human user to automatically build applications adapted to their context [26]. OCE learns online, that is, continuously from dynamically appearing data that is not known in advance [20], by reinforcement [23], and interactively [7]. OCE proposes these applications to the user who ultimately decides whether to accept them or not. Thus, the decision is shared between the human and the learning machine. Additionally, the user in the loop provides feedback on each proposed application, from which OCE automatically learns and builds a model usable for future decisions. Therefore, OCE and the human user form an hybrid intelligent system.

A prototype of an opportunistic composition engine has been tried out. It does indeed propose applications to the user. However, to verify that the learning solution works in a useful and trustworthy way, it is necessary to evaluate the quality of its propositions, i.e., their adaptation to the human user and the ambient environment. To achieve this, we are exploring a testing-based approach:

the problem is to test an online machine learning solution with the user in the loop. One of the challenges is the continuous presence of the human user in the interaction and decision loop, which cannot be maintained in a large-scale and repeatable testing process.

Testing machine learning-based solutions is a recent research field. Most of the work focuses on offline supervised learning and does not encompass all machine learning paradigms, particularly online learning [20] or interactive learning [7].

In this paper, we highlight the difference between testing models built by machine learning and testing the learning program that builds the models. **We target the learning program**, and we analyze the issues of testing in the context of online and interactive learning. We propose an approach based on the notion of **scenario** to **design**, **implement**, and **execute** test cases. We have developed two tools, Maker and Runner, which allow the implementation and execution of test scenarios coupled with OCE.

This paper is organized as follows. Section 2 presents the principles of opportunistic software composition. Section 3 outlines the main challenges in testing an online learning program with the human in the loop, based on key works in the literature. Section 4 introduces our scenario-based testing approach, then presents two tools for implementing (Maker) and executing (Runner) scenarios for evaluation purposes and describes their use and integration with OCE. Section 5 discusses how other works use the notion of scenario to test software and references some tools for developing ML-based solutions. Finally, Section 6 summarizes our contribution and its limitations, and discusses some perspectives.

2 OCE, an Online Learning Program with the User in the Loop

A software component is an executable software entity that provides services and requires others to function [21]. Component-based programming involves assembling components by connecting their services to form an application. The Figure 1 represents in UML [17] an example of a text-to-speech application that assembles four components: **TextInput** for text input, **TextToVoice** for text-to-voice conversion, a **Speaker**, and a **Button**. **TextInput** uses the *TextProcess* service provided by the **TextToVoice** component to transmit the text. **TextToVoice** converts the text into an audio signal and passes it to the **Speaker** component via the *VoiceProcess* service. The **Speaker** component plays the audio signal, which volume can be adjusted using the **Button** and transmitted to the **Speaker** via the *SetVolume* service.

OCE is a hybrid ambient intelligence solution that dynamically builds applications based on software components present in the ambient environment and knowledge learned about the user's preferences [26]. OCE automates the assembly operation. In ambient environments, characterized by their open and dynamic nature, components may appear or disappear unpredictably. Additionally, the user's needs and preferences can vary over time or depending on the current situation. A major challenge is therefore to manage the variability of

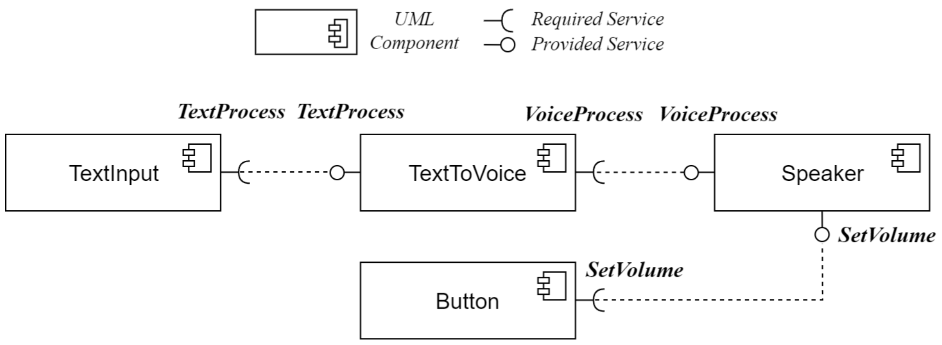

Fig. 1. Text-to-speech application represented in UML [17]

these environments and to offer the user "right" applications according to their situation.

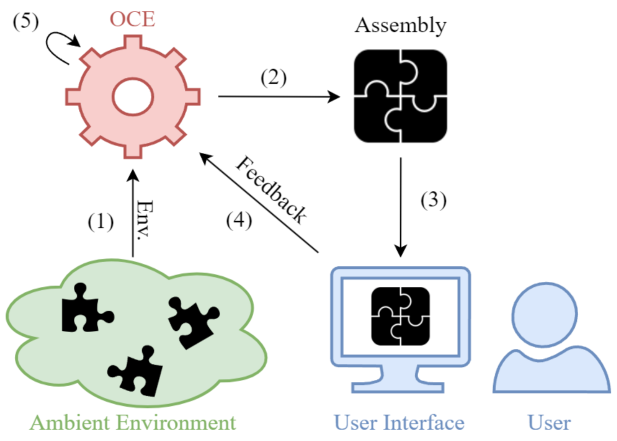

Fig. 2. OCE application building process

A "cycle" of OCE builds an application through several steps (Figure 2):

1. OCE detects the available components in the user's ambient environment.
2. Based on the available components and knowledge about the user's needs and preferences, OCE **decides** on the assembly to build.
3. OCE presents the assembly proposition to the user through a graphical interface.
4. The user accepts, modifies, or rejects the assembly proposition. This action translates into a reward, which serves as the source for OCE's learning.

5. OCE learns by reinforcement.

OCE repeats application building cycles. Thus, through successive cycles and interactions with the user, OCE learning solution builds and evolves a model on which the future decisions will be made.

A working prototype of OCE has been developed. Does it propose applications to the user that are tailored to their preferences and needs in the current situation? Testing is a way to answer this question and to build trust in OCE. However, testing software based on machine learning, especially when the user is at the center of the process, presents challenges that we analyze in the next section.

3 Machine Learning based Software Testing

Testing is an activity within the software development process [2, 9] that aims to uncover defects[1], compare different versions, tune parameters, or convince stakeholders that the software meets requirements.

Testing consists in running the software in various cases, in an environment close to the production environment, and then analyzing the results. Within the development team, it is the “oracle” that evaluates and interprets these results. If they indicate that the program does not behave as expected, the development team investigates the presence of a defect, returns to development to fix it, and then conducts another round of testing.

Testing machine learning solutions poses particular challenges, especially when the program learns online and interacts with humans, as is the case with OCE.

3.1 Problem Analysis

Unlike research conducted in the field of traditional software testing, research on testing machine learning-based software is relatively recent. Several papers address the general problem of developing machine learning-based software from a software engineering perspective but provide only brief explanations on testing issues [8, 14].

However, a few significant papers have been published in recent years. Zhang et al. [27] provided a comprehensive study on testing machine learning-based software, focusing on model evaluation and properties such as accuracy, robustness, and fairness.

Riccio et al. [19] systematically analyzed the literature and highlighted the main challenges related to testing, including test case specification, adequacy criteria, cost, and the oracle problem.

In the following, we highlight the difference between testing models and testing learning programs. Then, we focus on testing online learning in interaction with human users.

[1] A defect (or bug) is defined as an imperfection or flaw in a software product that does not meet the requirements or specifications.

Testing models *vs.* testing learning programs Machine learning encompasses learning activities, i.e., building a model by a learning program, and decision-making activities, i.e., using the model, whether these activities are interleaved or not. Models and learning programs are distinct artifacts, each of them requiring testing.

Model testing Before deploying a model, testing it involves answering the question: is the model a "good" model? In other words, did the machine "learn" effectively? Model testing shares the same goals as traditional software testing but focuses on quality properties specific to the model such as accuracy, relevance, or robustness [27]. This poses several challenges that we examine below.

Since models result from running a learning program fed by examples, defects may arise from the training data, the learning program, or a mismatch between the two (when a program poorly learns from certain data) [27]. However, it can be challenging to trace the source of a defect to correct it. Indeed, models do not have the same materiality as traditional software: they do not consist of a simple source code but are composed of more or less tangible various elements (code, parameters, data) and often operate as a "black box". In practice, focusing on specific properties can help locate the sources of defects: for example, testing the relevance of the model (such as overfitting [10]) can help identify defects in the training data.

On the other hand, machine learning is sometimes used by development teams in situations where the expected results are not known in advance [15]. In this case, predicting and interpreting test results is an additional challenge, and it can be difficult to determine whether a test passes or not. Software exhibiting this problem, known as the oracle problem, is often considered non-testable [25] due to the absence of an oracle or the difficulty in designing one [16].

Learning program testing Although the two problems are closely related, testing a learning program is not the same as testing a model. The question is: does the learning program "learn well" across the entire scope for which it was designed? In other words, does the learning program build "good" models relative to the data provided to it?

To answer this question, testers need to have different models built by the learning program for different use cases, with the aim of subsequently testing each of these models. The selection of training data to build these models is crucial for the overall significance of the tests. Moreover, since it is not possible to specify expected models for comparisons (the oracle problem again), each model must be tested to indirectly assess the quality of the learning program. Testing learning programs is therefore costly; it requires significant expertise and, as much as possible, automation.

In this work, we target the problem of testing learning programs.

Testing online learning programs in interaction with humans In the case of online learning in interaction with humans, testing presents particular challenges that we examine here.

In general, the quality of a model depends on the quality of the training data. In the case of OCE, this cannot be controlled due to the unpredictable dynamics of the ambient environment, as well as because the user needs, expectations, or preferences may vary over time and depending on the situation. The user may also misunderstand the model's outputs and so provide irrelevant feedback. In such cases, the model may produce test results that are deemed incorrect even though the learning mechanism is working properly. Therefore, it is challenging to determine whether the machine failed to learn or if the issue arises from interactions between the user and the learning program. Furthermore, the quality of the results produced by a model may depend on users: they may be good for one user but not for another. More generally, it is difficult to define test cases that encompass a wide range of user profiles while anticipating their dynamics or possible inconsistencies.

Under these conditions, the question of testing does not concern the presence of a defect in the training data; rather, it involves verifying the consistency between the training data and the model built from that data. Thus, a test case must first construct a model in interaction with a user and then evaluate it, which makes both the design and the execution of the test more complex.

Moreover, the potentially non-deterministic nature of machine learning [13, 22], particularly (but not exclusively) in the case of online learning, must be taken into account. The randomness part inherent in learning and decision-making mechanisms leads to variable outcomes from one execution to another, even with the same configuration and inputs. Thus, it can happen that the model does not produce expected results during tests even though the learning and decision-making mechanisms are working correctly [12]. For example, in reinforcement learning (often performed online), it is normal for the machine to sometimes choose, for exploration purposes, a solution that is neither the best nor the logically expected one. Therefore, it is difficult to determine if an unexpected output results from a random factor or a flaw in the learning mechanism.

Another issue concerning the test process management lies in the continuous involvement of humans in the learning and decision loop: during the development phase of the learning program, it is not feasible to engage human users (or testers fulfilling their role) in order to test and repeat a significant number of cases. Therefore, conducting the test process, it is necessary to "automate" the interaction with humans.

Summary Below, we outline the research questions we have identified regarding the automated testing of learning programs. The first two relate to the design and implementation of test cases, while the latter two concern their execution and analysis:

– **RQ1.1**: How to design a test case that includes learning and evaluation steps, while integrating human interaction?
– **RQ1.2**: How to implement a test case with the purpose of automating its execution?
– **RQ2.1**: How to automate test execution and consider non-determinism?

– **RQ2.2**: How to measure the quality of obtained results, i.e., determine if the test is passed or not?

The following section provides answers to these questions and presents a solution for designing, implementing, and executing test cases for assessing OCE. There are other approaches that can address these questions: for example, metamorphic testing [5] focuses on verifying model behavior in the absence of an oracle and targets question **RQ2.2**. Furthermore, other properties are required for an interactive online learning program, related to human factors and user experience (usability, cognitive load, engagement, etc.), but these questions are beyond the scope of this paper.

4 A Scenario-based Approach to Evaluate OCE

OCE is a program that learns online with the user in the loop. To test it, we must evaluate the different models it builds and it updates from one cycle to the next: are these models capable of proposing relevant applications for the user, i.e., in line with their preferences and situation? Since a model cannot be compared to an expected model and is not suitable for code analysis, it must be executed to verify the quality of its outputs, i.e., the ambient applications it proposes. It is important to note that the aim is not to evaluate the raw quality of these applications (functionality, performance, security, etc.), but rather whether they meet the user's expectations and habits, precisely their compliance with what the user has previously expressed.

This section first examines what test cases are and their structure. It then presents two tools, Maker and Runner, which enable the implementation and automatic execution of these test cases coupled with OCE.

4.1 Test Scenario

In an iterative context such as online learning, designing a test case requires defining a sequence of interactions between the learning program and its learning environment (the ambient environment and the user in our case). Online learning requires a certain number of interactions to learn and derive a model. To verify that the model behaves as expected, additional interactions are necessary. We call this sequence of interactions dedicated to learning and evaluation a "test scenario" [2]. The interactions related to learning and evaluation can be intertwined. However, in this paper and for the sake of simplicity, we consider only one learning phase and one evaluation phase executed sequentially.

The **learning phase** involves a sequence of cycles where, for each cycle, OCE makes a proposition, the user provides feedback, and then OCE learns

[2] In the field of software testing, the term "test scenario" usually refers to the organization and planning of the testing process within a development project. Our definition of the term test scenario is different: it is a usage scenario intended to be tested, thus constituting a test case.

from the feedback (as depicted in Figure 2). To define the learning phase, each cycle must be specified by: *(i)* the list of software components populating the ambient environment (which changes from cycle to cycle) along with their services for assembly, and *(ii)* to avoid the tester having to constantly interact with OCE during tests, which would be too tedious and costly, the "ideal assembly" specifies the outcome (output) that the user would expect in this case. Based on the assembly of components proposed by OCE and the ideal assembly, OCE generates feedback and learns, just as it does in normal operation. At the end of this phase, OCE has built a model adapted to the environmental configurations (including user feedback). The model is then ready for evaluation.

To simplify the tests and the analysis of the results, the **evaluation phase** can be reduced to a single cycle, but this is not mandatory. An evaluation cycle is defined by: *(i)* the ambient environment as in the learning phase and *(ii)* the expected output called the "expected assembly". Thus, by providing the expected output, the test designer acts as an oracle. Distance measures between the output proposed by OCE and the expected output are calculated to evaluate the relevance (correctness in the sense of Zhang *et al.* [27]) of OCE's decision.

This scenario-based approach addresses the research question **RQ1.1**. In this way, it is possible to define and test in different ambient environments and their dynamics, as well as different user behaviors. It should be noted that issues related to identifying representative test scenarios and involving the human (the user or their representative) in their definition are beyond the scope of this paper.

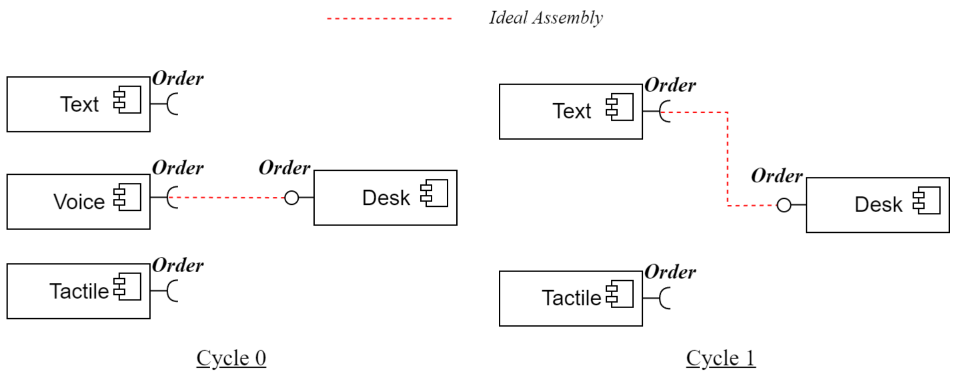

Fig. 3. Learning cycles of the example scenario

Example To illustrate what a scenario is, let's take an example, adapted from [26] and simplified for clarity. Mary is at work. In her ambient environment, there are a software component **Desk** that provides a room booking service called *Order* and three components, **Text**, **Voice**, and **Tactile**, which allow the user to make a booking request (with different interaction modes) and require the *Order* service. Through the *Order* service, these three components can be assembled with **Desk**. Thus, three applications are possible (**Text-Desk**, **Voice-Desk**, **Tactile-Desk**), enabling Mary to book a meeting room. As testers, we want to verify that if Mary expresses a preference for **Voice-Desk**, then when a similar situation arises, OCE will again propose **Voice-Desk**.

Let's imagine a simple scenario with three cycles where **Voice** disappears and then reappears. Cycles 0 and 1 (Figure 3) define the learning phase. Cycle 0 is defined by the list of 4 components and the ideal assembly **Voice-Desk** (Mary's preference). For cycle 1, the components are **Desk**, **Text**, and **Tactile**. The ideal assembly in this case is **Text-Desk**.

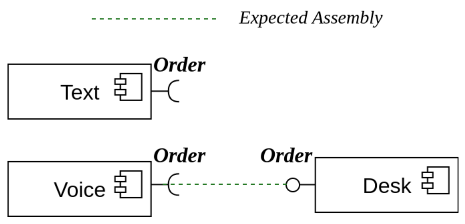

Fig. 4. Evaluation cycle of the example scenario

In the evaluation phase, cycle 2 (Figure 4) is defined by the list composed of **Desk**, **Text**, and **Voice**, and the expected assembly is **Voice-Desk**. Next, we present two tools that support the definition and execution of such a scenario.

4.2 Tools for Testing OCE

The tools we have developed to test the learning program OCE are presented in this section. The source code is available[3], along with a short video[4] that complements this section and demonstrates their usage in the scenario described above.

Maker This is an interactive tool for implementing scenarios that addresses research question **RQ1.2**. For a given cycle, the tester can reuse or define fictitious components (Figure 5, left panel), drag and drop them into a panel defining the environment, and link their services to define the ideal or expected assembly (figure 5, right panel). Additional features, such as the ability to duplicate a cycle, reduce the tester's workload. From a sequence of cycles, Maker generates a JSON file that implements the scenario.

[3] `https://www.irit.fr/OppoCompo/resources/`
[4] `https://www.irit.fr/OppoCompo/makerrunnerusecase2024/`

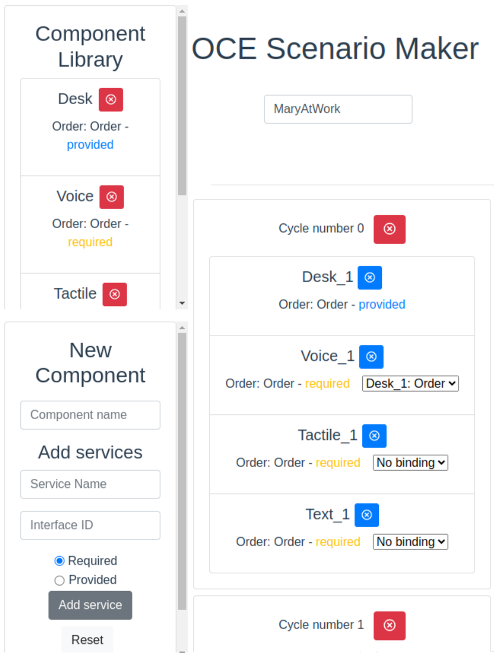

Fig. 5. Maker's interface for scenario implementation

Runner This is a Java application that, coupled with OCE, allows the execution of scenarios in JSON format, such as those generated by Maker. To mitigate the impact of the non-deterministic nature of machine learning, a scenario can be repeated multiple times, and average values of the measured results are calculated. Several parameters need to be defined, such as learning hyperparameters (e.g., exploration rate of reinforcement learning), the version of OCE to test, and the number of repetitions. Thus, Runner addresses research question **RQ2.1**. Once the parameter values are set, it operates coupled with OCE without further intervention from the tester.

To answer the research question **RQ2.2** and assess the relevance of the applications proposed during the evaluation cycles, Runner compares the assembly proposed by OCE with the expected assembly by calculating a Jaccard similarity index[5]. It provides an average index across all evaluation cycles of a scenario. This measure indicates whether the constructed models have made relevant propositions based on what OCE learned, i.e., based on the user's preferences in different ambient environments.

Architecture and Implementation Maker and Runner have been implemented and integrated into the OCE system, which architecture can be represented in the form of a UML component diagram too [17]. Figure 6 represents

[5] `https://en.wikipedia.org/wiki/Jaccard_index`

the configuration of the OCE system in production. When the ambient environment changes, **OCE** receives from **Ambient Env.** the list of components that are currently present (service *ProcessOneEnv*). **OCE** builds an assembly and proposes it to the user via the **User Interface**, and the user provides feedback in the form of an assembly (service *Feedback*).

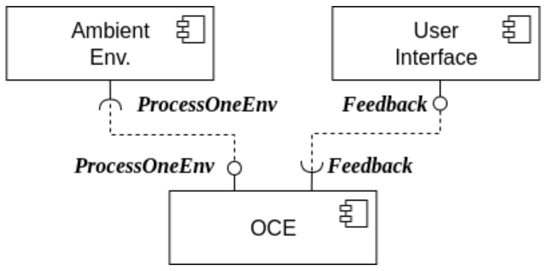

Fig. 6. Architectural view of OCE in production configuration

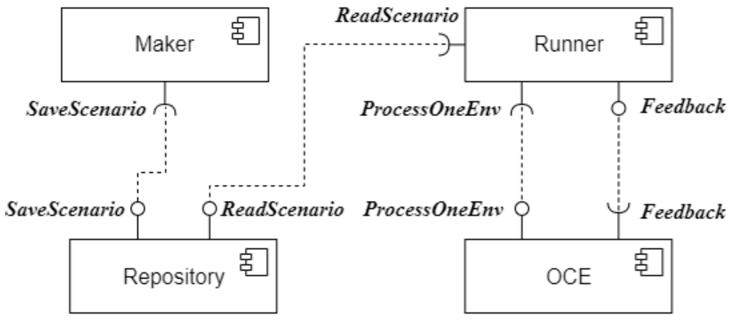

Fig. 7. Architectural view of OCE in testing configuration

Figure 7 shows how Maker and Runner have been integrated into OCE architecture. The modularity of this architecture allows replacing the ambient environment and the user interface with Runner to perform tests implemented with Maker. In the testing configuration, **Runner** executes a scenario taken from the **Repository** of scenarios (service *ReadScenario*). The directory is populated by scenarios implemented with **Maker** (service *SaveScenario*). For each cycle of the scenario, **Runner** requests **OCE** to propose an assembly (service *ProcessOneEnv*). Following the proposition, **Runner** provides **OCE** with the ideal assembly (service *Feedback*), which **OCE** treats as user feedback. In the case of evaluation cycles, **Runner** computes the distance between the proposed assembly and the ideal one defined in the scenario.

Experimentation Maker and Runner were used during an initial test campaign of OCE's behavior. We defined around ten scenarios aimed at testing basic cases, such as learning a preferred component, learning to avoid a component, or deal with sensitivity to novelty (unknown components that appear in the ambient environment). For each case, we varied the dynamics of the environment. Runner was tested with scenarios created using Maker. No failures were observed with these two tools. They simplified the tester's work and accelerated the campaign by avoiding the need to manually run OCE hundreds of times. They enabled triggering changes in the environment to simulate its dynamics and provided a relevance measure without laborious analysis of OCE's knowledge.

These tests helped uncover a few flaws in OCE's decision-making process. They also enabled an initial evaluation of the learning process, refinement of parameter settings, and correction of defects in the handling of the user feedback.

5 Related Work

This section positions our approach in relation to other works using the concept of scenario and a few tools for developing and experimenting ML-based solutions.

5.1 Scenarios for Testing

C. Kaner [4] defines a scenario as a story of a person trying to accomplish something with the tested product. Hussain et al. [11] propose a similar definition, but not directly tied to testing: a scenario is an informal description of a specific use of software, or of a part of it, by a user. Here, scenarios are defined from user needs (use cases) and are used to derive test cases. According to these definitions, a scenario is described as a sequence of interactions between the software and a user. This is precisely how we use scenarios to test OCE (cf. Section 4). Indeed, a scenario describes a sequence of ambient environments (where an environment is represented by a list of software components) as well as ideal assemblies that model interactions between the user and OCE.

In the domain of autonomous vehicles, the concept of scenario is used to evaluate vehicle behaviors based on machine learning. For instance, Ulbrich et al. [24] describe a scenario as a temporal sequence of scenes, where each scene represents a configuration of the physical environment in which one or more autonomous vehicles operate. A scenario details the scenes, the transitions between different scenes, and the evaluation criteria. In this context, a scenario specifies sequences of physical environments rather than user-software interactions.

The scenario-based approach enables the comprehensive description of a use case, thereby facilitating end-to-end testing ("system" level testing). This makes it particularly suitable for evaluating machine learning-based software due to its "black box" nature. We have tailored this approach for "online" learning by interleaving learning and evaluation phases within a scenario. Furthermore, we have adapted it for "interactive" learning by allowing the expression of interactions with the user within a scenario.

5.2 Some Tools

Tools like Gymnasium [3], DotRL [18], and Cogment [1] are designed for the development and experimentation of reinforcement learning solutions. Evaluation of a solution typically involves comparing it against other algorithms in predefined environments. However, these tools do not include the notion of scenarios. Additionally, unlike Runner, they do not provide measures to simplify the analysis of test results. They often supply raw data, leaving their interpretation to the tester.

Cogment is the only platform that integrates the human in the interaction loop, focusing on interactive learning. This platform facilitates the development of solutions where humans can intervene to accelerate learning. However, its current focus is on solution deployment rather than providing testing facilities.

6 Synthesis and Perspectives

To meet the evaluation needs of an online learning solution with human interaction, we analyzed the testing challenges in the context of machine learning, focusing on testing interactive online learning. To address our research questions regarding design, implementation, and execution of tests of a learning program interacting with humans, we proposed an approach based on scenarios that mixes learning and evaluation phases. To implement this approach and apply it to the Opportunistic Composition Engine (OCE), we introduced two tools: Maker and Runner. These tools are operational but currently evolving. Several new features are planned, such as allowing testers to define their own quality metrics (e.g., precision or recall) or providing the ability to monitor scenario execution cycle by cycle.

Although defined to meet the evaluation needs of OCE, we believe that the principles we propose can be helpful for testing other online learning solutions with the human in the loop.

Our proposition still leaves several questions regarding the involvement of the human user in the testing process and the management of a test campaign. Thanks to the scenario-based approach, the presence of the user is no longer necessary during test execution. However, the contribution of users (or their representatives, experts of the domain) is still necessary in the design of scenarios. On the other hand, to cover the scope of application of the learning program effectively, it is necessary to have a potentially large number of test scenarios. Typically, this number is too large for scenarios to be designed one by one "by hand". The issue then revolves around automatically generating test scenarios: how to automatically generate sequences of environments with their dynamics? How to automatically integrate the interaction with the user and their feedback? To scale up the number of scenarios, it is necessary to replace the user with an "automaton" that models a typical behavior and provides ideal assemblies. Additionally, how to manage the combinatorial explosion and select test scenarios (knowing that it is not possible to test a program exhaustively)?

Our aim is to mix models of ambient environments and user behaviors models to derive test scenarios, similar to those produced by Maker, in order to execute them using Runner.

References

1. AI Redefined, Gottipati, S.K., Kurandwad, S., Mars, C., Szriftgiser, G., Chabot, F.: Cogment: Open Source Framework For Distributed Multi-actor Training, Deployment & Operations. CoRR **abs/2106.11345** (2021), `https://arxiv.org/abs/2106.11345`
2. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2016)
3. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. CoRR (2016), `https://arxiv.org/abs/1606.01540`
4. Cem Kaner, J.: An introduction to scenario testing. Florida Institute of Technology, Melbourne pp. 1–13 (2013)
5. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. arXiv preprint arXiv:2002.12543 (2020)
6. Dunne, R., Morris, T., Harper, S.: A survey of ambient intelligence. ACM Computing Surveys (CSUR) **54**(4), 1–27 (2021)
7. Fails, J.A., Olsen Jr, D.R.: Interactive machine learning. In: Proceedings of the 8th Int. Conf. on Intelligent User Interfaces. pp. 39–45 (2003)
8. Giray, G.: A software engineering perspective on engineering machine learning systems: State of the art and challenges. J. of Systems and Software **180**, 111031 (2021)
9. Graham, D., Black, R., van Veenendaal, E.: Foundations of Software Testing: ISTQB Certification. Cengage, 4 edn. (2020)
10. Hawkins, D.M.: The problem of overfitting. Journal of chemical information and computer sciences **44**(1), 1–12 (2004)
11. Hussain, A., Nadeem, A., Ikram, M.T.: Review on formalizing use cases and scenarios: Scenario based testing. In: 2015 Int. Conf. on Emerging Technologies (ICET). pp. 1–6. IEEE (2015)
12. Khomh, F., Adams, B., Cheng, J., Fokaefs, M., Antoniol, G.: Software engineering for machine-learning applications: The road ahead. IEEE Software **35**(5), 81–84 (2018)
13. Marijan, D., Gotlieb, A., Ahuja, M.K.: Challenges of Testing Machine Learning Based Systems. In: Proc. of the 1st IEEE Artificial Intelligence Testing Conf. IEEE, San Francisco, CA, USA (2019)
14. Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A.M., Wagner, S.: Software engineering for AI-based systems: a survey. ACM Trans. on Software Engineering and Methodology (TOSEM) **31**(2), 1–59 (2022)
15. Murphy, C., Kaiser, G.E., Arias, M.: An approach to software testing of machine learning applications. In: Int. Conf. on Software Engineering and Knowledge Engineering (2007)
16. Nakajima, S.: Generalized Oracle for Testing Machine Learning Computer Programs. In: Software Engineering and Formal Methods - SEFM 2017. LNCS, vol. 10729, pp. 174–179. Springer (2017)

17. OMG: Unified Modeling Language, chap. 11.6 (2017), `https://www.omg.org/spec/UML/2.5.1/PDF`
18. Papis, B., Wawrzyński, P.: dotRL: A platform for rapid Reinforcement Learning methods development and validation. In: 2013 Fed. Conf. on Computer Science and Information Systems (FEDCSIS). pp. 129–136. IEEE (2013)
19. Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., Tonella, P.: Testing machine learning based systems: a systematic mapping. Empirical Software Engineering **25**, 5193–5254 (2020)
20. Russell, S.J., Norvig, P.: Artificial intelligence: A Modern Approach. Pearson Education, Inc. (2010)
21. Sommerville, I.: Component-based software engineering. In: Software Engineering, pp. 464–489. Pearson Education, 10^{th} edn. (2016)
22. Sugali, K.: Software testing: Issues and challenges of artificial intelligence & machine learning. Int. J. of Artificial Intelligence & Applications **12**(1), 101–112 (2021)
23. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press, 2nd edn. (2018)
24. Ulbrich, S., Menzel, T., Reschka, A., Schuldt, F., Maurer, M.: Defining and substantiating the terms scene, situation, and scenario for automated driving. In: IEEE 18th Int. Conf. on intelligent transportation systems. pp. 982–988. IEEE (2015)
25. Weyuker, E.J.: On testing non-testable programs. The Computer Journal **25**(4), 465–470 (1982)
26. Younes, W., Trouilhet, S., Adreit, F., Arcangeli, J.P.: Agent-mediated application emergence through reinforcement learning from user feedback. In: 29th IEEE Int. Conf. on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). pp. 3–8. IEEE Press (2020)
27. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Trans. on Software Engineering **48**(1), 1–36 (2020)