Scenario-Based Testing of Online Learning Programs

Maxence Demougeot[1], Sylvie Trouilhet[1], Jean-Paul Arcangeli[1] and Françoise Adreit[2]
[1]*IRIT, Université de Toulouse, Toulouse, France*
[2]*IRIT, Université de Toulouse, UT2J, Toulouse, France*

Keywords: Online Machine Learning, Learning Program, Test, Test Scenario, Functional Testing, Testing Process.

Abstract: Testing is a solution for verification and validation of systems based on Machine Learning (ML). This paper focuses on testing functional requirements of programs that learn online. Online learning programs build and update ML models throughout their execution. Testing allows domain experts to measure how well they work, identify favorable or unfavorable use cases, compare different versions or settings, or reveal defects. Testing programs which learn online has particular features. To deal with them, a scenario-based approach and a testing process are defined. This solution is implemented and automates test execution and quality measurements. It is applied to a program that learns online the end-user's preferences in an ambient environment, confirming the viability of the approach.

1 INTRODUCTION

Machine Learning (ML) techniques (Mitchell, 1997) can be used when a software solution cannot be programmed, either because there is no known algorithm, or because the algorithm is too complex to implement. ML relies on a learning algorithm which processes reference examples called training data, from which it generalizes a behavior to hold (this may be predicting, deciding, executing a task…) when faced with data other than training data. The **learning program** (LP) is the program that implements the learning algorithm, and a **model** is a product of running the learning program with training data.

ML therefore consists in building (training) a model by means of a learning program. In production, the model transforms input data into outputs (i.e., predictions, decisions…) in accordance with the behavior learned from the training data. In Offline Machine Learning, the model is built and tuned by one or more human experts (ML experts and domain experts). They are responsible for choosing an off-the-shelf learning component (i.e., a learning program) and setting the learning parameters (hyperparameters). Alternatively, a specific learning program can be developed with the help of a software expert. ML and domain experts also select and prepare the training data. After construction and fine-tuning, the model is put in production. It can be rebuilt, i.e., revised or adapted, depending on the results obtained in production, however the rebuilt and production phases are separate. The model is thus managed like standard software with maintenance steps.

Sometimes, it is difficult to anticipate the data that the model will face in production, and the model needs to adapt over time according to the data encountered. **Online Machine Learning** addresses this issue. According to S. Russell and P. Norvig (Russell and Norvig, 2010), online ML relies on repeated comparisons between the outputs delivered in production and what they should be (the outputs assumed to be the "right" ones): when the model produces an output for a given input, a domain expert provides it with the "right" one as a feedback, triggering a new learning phase. In that case, learning and production are intertwined, and learning is continuous: the learning program updates the model iteratively and incrementally throughout its execution. Since, the learning context may be open and unpredictable, it has to deal with unanticipated training data arriving on the fly, which may be of poor quality but whose defects cannot be corrected before learning as with offline learning.

In ML, testing is part of the model development and tuning process, with the aim of assessing its quality, especially its compliance with functional requirements. Upstream, the LP must also be tested as defects can lead to the production of poor models. Section 2 analyzes these issues with a particular focus on online learning, and highlights the differences between testing a model and testing a learning program.

Demougeot, M., Trouilhet, S., Arcangeli, J.-P. and Adreit, F.
Scenario-Based Testing of Online Learning Programs.
DOI: 10.5220/0013503100003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 99-110
ISBN: 978-989-758-757-3; ISSN: 2184-2833

This paper addresses the problem of **testing learning programs** in the context of online ML. It focuses on **functional testing**: the aim is to verify that the machine learns "well", i.e., that the LP builds models whose behaviors are in line with the training data in the various learning situations it may face. Testing is a way to show how well the LP works, to identify favorable or unfavorable use cases, to compare different versions or settings, or to reveal defects.

To solve this problem, we propose a **scenario-based approach** and a **testing process**. Test scenarios are designed by **domain experts** without necessarily skills in ML, based on properties to be controlled. Then, they are automatically transformed and run to produce quality measurements. We apply this solution to a program called OCE, which learns online how to build applications for an end-user in an open context. We show how our approach can be used, present a toolset that implements our proposal within the OCE framework, and examine several use cases that show the viability of our solution.

The paper is organized as follows. Section 2 analyzes the problem of testing learning programs and sets out the research questions we are addressing. Section 3 introduces our case study: the opportunistic composition engine (OCE). Section 4 describes the scenario-based approach to test programs that learn online, with a process for carrying it out. Section 5 presents a solution to implement and run scenarios, with a toolset we have developed to test OCE's learning program. This approach has been validated in a number of use cases, some of which are presented in Section 6. Section 7 studies the related work. Finally, we draw some conclusions in Section 8.

2 LP TESTING

Work on ML-based software testing mainly focuses on models resulting from offline supervised learning (Zhang et al., 2020). In this work, we target learning programs that build models online and iteratively. In the following, we highlight the differences between testing models and testing learning programs, and focus on testing programs that learn online.

2.1 LP Testing vs Model Testing

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

A common problem is the lack of precise specifications, which limits verification possibilities. Another is the **non-deterministic nature of machine learning** (Sugali, 2021), particularly (but not exclusively) in the case of online learning. Randomness inherent in learning and decision-making mechanisms leads to variable outcomes from one execution to another, even with the same inputs. Thus, it can happen that the expected results are not obtained during tests even though the learning and decision-making mechanisms are working correctly (Khomh et al., 2018). For example, in reinforcement learning (Sutton and Barto, 2018), 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 defect in the learning mechanism.

Model Testing: Before deploying a model, testing aims to answer 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 (Zhang et al., 2020). It consists in **running the model** and **evaluate the decisions** it makes. 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) (Zhang et al., 2020). However, it is 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".

On the other hand, ML is sometimes used by development teams in situations where the expected results are not known in advance (Murphy et al., 2007). 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(Weyuker, 1982) due to the absence of an oracle or the difficulty in designing one (Nakajima, 2017).

Interpreting the results can also be tricky, due to the nature of the outputs and their complexity. In this context, it is so challenging to measure the quality of models using commons criteria such as precision (ratio of correct answers to given answers), recall (selec-

tion rate of correct answers), or F-scores (combination of precision and recall).

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 scope for which it was designed? In other words, does the learning program build "good" models relative to the training data provided to it?

The **correspondence between training data and model** is captured by the formula 1, in which D is a training dataset, *Scope* is the set of all training datasets that the LP might encounter, $P1$ is a property of D, M_D is the model built by the LP from D, and $P2$ is an expected property of M_D:

$$\forall D \in Scope, P1(D) \implies P2(M_D) \quad (1)$$

In other words, this formula means that if the learning program has built a model M_D from training data D that satisfy $P1$, then M_D should satisfy $P2$. For illustration purposes, let's take the example of a robot that learns to move from one point to another in a complex and dynamic physical space. In this case, the LP builds a model to guide the robot from reference routes. We might want to check that **IF** *D contains no routes that use a given corridor C* ($P1$) **THEN** *to go from point A to point B, M_D guides the robot without passing through corridor C* ($P2$). As usual in testing, the goal is to assess (un)satisfaction of the formula 2, that captures the **mismatch between training data and model**, i.e., a poor transformation of training data into a model:

$$\exists D \in Scope, P1(D) \wedge \neg(P2(M_D)) \quad (2)$$

However, identifying $P1$ as well as selecting D is crucial to the overall significance of the tests and demands strong domain expertise. Neither D nor M_D are simple data, as numerical data for example. In the case of online ML, D is a **sequence** of training data which is required to meet $P1$. Selecting D is all the more delicate since, in production, data arrives in sequence and are not always expected. Hence, the question is not about the presence of a defect in the training data (a common source of defects in models); rather, it's about checking the consistency between training data and the model built from them (possibly to conclude that there is a mismatch between the two).

Defining an expected property of the model ($P2$) is another source of complexity: especially, it is not possible to specify expected models for comparisons and accuracy assessment (the oracle problem again). $P2$ is thus a property related to the decisions that a model makes. To check if the latter satisfies $P2$, and thus indirectly evaluate the quality of the learning program, it must be executed. Therefore, LP testing suffers from the problems associated to model testing.

Cross-Validation and K-fold cross-validation (Berrar, 2019) are standard methods for testing the performance of models, especially overfitting and underfitting. They are commonly used for comparison and selection of the most appropriate model. They consist in putting aside part of the training data to use it as test data. But, in the context of online learning, they are not applicable for assessing the correspondence between the sequence of training data and the model since they do not allow to capture the consequence relationship between data and model expessed by the IF-THEN formula 1.

Thus, to tackle the LP testing problem, 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. So, running a test case must **first construct a model then run and evaluate it**. This makes both design and execution of the test cases more complex.

Testing learning programs is therefore costly; it requires significant **expertise** in terms of domain and business and, as much as possible, **automation**.

2.2 Research Questions

Based on the above analysis, we have identified three research questions regarding the testing of online learning programs:

- **RQ1:** How to design test cases that include learning and evaluation steps and conform to expected properties?
- **RQ2:** How to organize the testing workflow, from the test set design to execution and evaluation?
- **RQ3:** As part of the workflow, how to implement a test case, automate its execution, and determine if the test passes or not?

Before presenting our answers to **RQ1** and **RQ2** in Section 4, and to **RQ3** in Section 5, Section 3 introduces the fundamental principles of our case study, OCE, which is a program that learns online in interaction with the end user.

3 OCE: A PROGRAM THAT LEARNS ONLINE

Let's start with some background to better understand what the Opportunistic Composition Engine (OCE)

does. A software component is an executable software entity that provides services and requires others to function (Sommerville, 2016). Component-based programming involves assembling components by binding their services to form an application. The Figure 1 represents in UML (OMG, 2017) a simple example of a text-to-speech application that assembles four components: **TextInput** for text input, **TextToVoice** for text-to-voice conversion, **Speaker**, and **Button**. **TextInput** uses the *TextProcess* service provided by **TextToVoice** to transmit the text. **TextToVoice** converts the text into an audio signal and passes it to **Speaker** via the *VoiceProcess* service. **Speaker** plays the audio signal, which volume can be adjusted using **Button** and transmitted to **Speaker** via the *SetVolume* service.

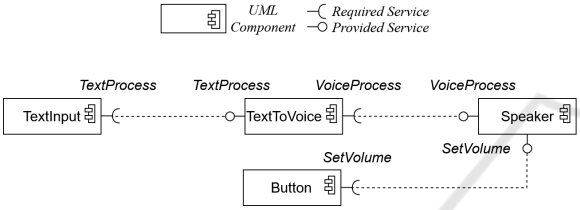

Figure 1: UML component diagram (OMG, 2017) of the Text-to-Speech application.

OCE is an ambient intelligence (Dunne et al., 2021) prototype solution that dynamically builds applications based on software components present in the ambient environment and knowledge learned about the user's preferences (Younes et al., 2020). OCE automates the assembly operation. In ambient environments, characterized by their open and dynamic nature, components may appear or disappear unpredictably. So a major challenge is to manage the variability of these environments and to offer the user "right" applications according to their situation.

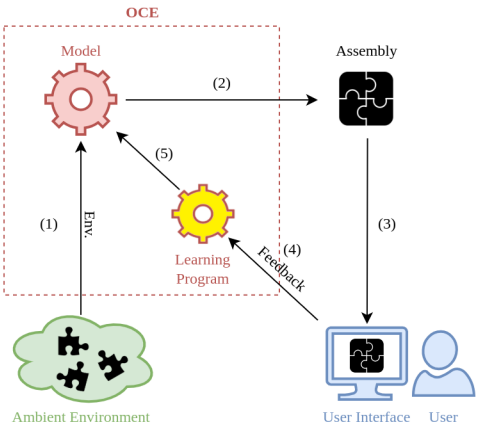

Figure 2: OCE application building process.

OCE consists of a reinforcement learning (Sutton and Barto, 2018) program and a model that is continuously trained and evolving. An OCE "cycle" is a sequence of several steps (Figure 2):

1. OCE detects the available components in the user's ambient environment.
2. Based on the available components and learned knowledge about the user's preferences, OCE's model 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. These actions translate into positive or negative rewards, which serve as training data for OCE's learning program.
5. OCE learns by reinforcement, i.e., OCE's learning program increments and adapts the model.

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

Several working prototypes of OCE have been developed. Do they propose applications to the user that are tailored to their preferences in the current situation? Testing is a way to answer this question and to build trust in OCE.

4 SCENARIO-BASED TESTING APPROACH

4.1 Test Scenario

To be evaluated, a learning program must be executed on training data. Then, the resulting model must be evaluated in turn. Since the learning program builds different models depending on the training data, different models must be evaluated. In an iterative and incremental context such as online learning, designing a test case requires defining a sequence of interactions between the learning program and its learning environment to build a model. To check that the built model behaves as expected, other interactions are required for evaluation purposes. Learning and evaluation interactions can be intertwined. We call a sequence of such interactions a **test scenario**.

A **learning phase** consists in a sequence of interactions which must conform to $P1$ with, for each interaction, an output produced by the model from the input data, a feedback, then a learning operation to improve the model. Each interaction is so described

by: (i) a learning situation, i.e., the environment in which the model should produce an output, (ii) an output that the model should ideally return in this situation, called **ideal** output. Defining the ideal output for each learning interaction, as well as expected outputs facilitates automation: it avoids the need for the tester to constantly interact with the learning program, which is tedious and costly. Based on the outputs produced by the model and the ideal outputs, the learning program learns and updates the model in the learning phase of the test as it does in normal operation. At the end of this phase, a model is built that is ready for evaluation.

An **evaluation phase** consists of one or several interactions too. An interaction is defined by the description of both the situation in which the model has to deliver an output and a property of this output that must be checked ($P2$). In its simplest form, the property can be the expected output or several possible correct outputs. Expected outputs are intended to be compared with the output returned by the model to give a measure of quality. Depending on what the designer expresses in terms of expected property (a single or several expected outputs, or a more general property), they act more or less like an oracle.

Organized in this way, such scenarios support the testing of online learning programs, which answers **RQ1**. We now apply this solution to a particular case: OCE's online learning program.

Application to OCE. For OCE, an interaction corresponds to a cycle, and a test scenario is a sequence of learning and evaluation cycles. A **learning cycle** is described by a list of software components populating the ambient environment and the ideal assembly. The latter specifies the assembly the user would accept in normal use in this situation, and which is used to learn. An **evaluation cycle** is also defined by a list of software components, this time with a property of the assembly that OCE's model is expected to return. This can be in the form of one or more expected assemblies for comparison, or take the form of more general properties, e.g., the presence of certain components or connections in the assembly.

It is possible to define scenarios to test OCE's learning program in different ambient environments, with more or less dynamics according to $P1$.

For instance, as a tester, we seek to verify that **IF** *a user expresses a preference for a component in a certain situation, and that component disappears from the ambient environment* ($P1$) **THEN** *if it reappears in a similar situation, OCE again proposes an assembly with that component* ($P2$). To define a scenario, let's take a toy example adapted from (Younes et al., 2020) 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 may be assembled with **Desk**. Thus, three applications are possible (**Text-Desk**, **Voice-Desk**, **Tactile-Desk**), enabling Mary to book a meeting room. To do so, Mary prefers the component **Voice**. Then, **Voice** disappears and reappears later. We seek to verify that OCE again proposes the assembly **Voice-Desk** when **Voice** reappears.

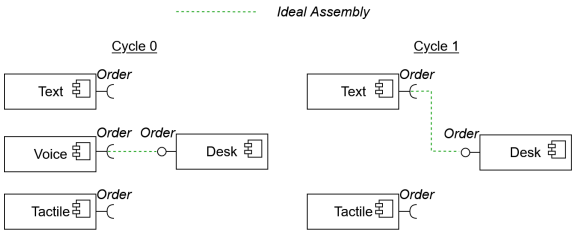

Figure 3: Learning cycles of the example scenario.

Let's define a scenario with three cycles. Cycles 0 and 1 (Figure 3) define the learning phase. Cycle 0 is defined by the list of the 4 components and the ideal assembly **Voice-Desk** (Mary's preference). For cycle 1, **Voice** has disappeared, so the components are **Desk**, **Text**, and **Tactile**, and the ideal assembly in this case is **Text-Desk**. The composition of the ambient environment and the ideal assemblies in cycles 0 and 1 satisfy $P1$.

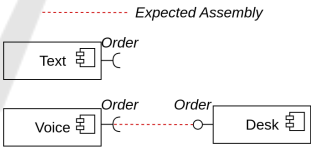

Figure 4: Evaluation cycle of the example scenario.

In the evaluation phase, cycle 2 (Figure 4) defines a situation, different from the two previous ones, in which OCE must provide an assembly: the ambient environment is composed of **Desk**, **Text**, and **Voice** that has reappeared (**Tactile** has desappeared). Here, **Voice-Desk** is specified as the expected assembly in order to be compared with the assembly proposed by OCE. Here, $P2$ is defined at the simplest by a single expected assembly and coupled with a function that computes a distance between the expected assembly and the OCE output. To measure this distance, such a function may, for instance, compute a Jaccard Similarity Index[1].

[1] https://en.wikipedia.org/wiki/Jaccard_index

4.2 Scenario-Based Testing Process

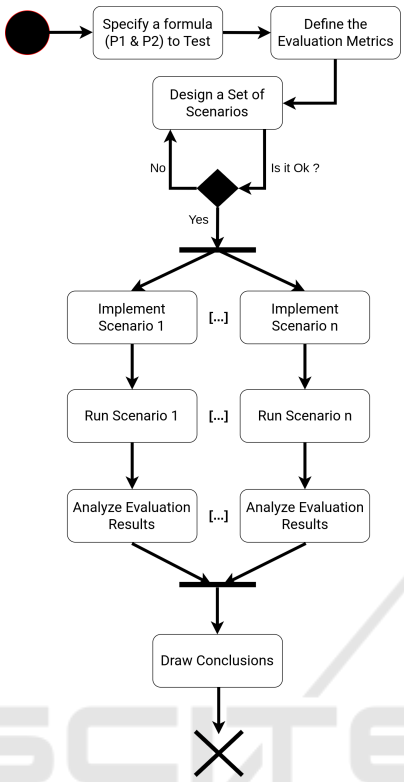

Figure 5: Activity Diagram of the Scenario-based Testing Process.

In a scenario-based testing process (Figure 5), the first activity is to identify a formula to be tested, namely P1 which describes a property of the training data (in the case of OCE, a property that the sequence of ambient environments and ideal outputs must satisfy), and P2 which relates to the expected result.

Next, it is necessary to define the ways of verifying that P2 is satisfied. A standard way is to use statistical metrics based on the performance of the model, such as accuracy, recall, or F1-score. In the case of OCE, a metric based on the distance between proposed assemblies and expected assemblies can be used to check whether the model has made good assembly proposals or not.

Then, an iterative design of the scenario test set begins. Defining scenarios is guided by tactics, e.g., limiting the complexity of the training sequence conforming to $P1$ in order to facilitate the analysis of test results. For each scenario, the learning phase is designed to meet $P1$ and the evaluation phase is designed to enable the assessment of $P2$.

Once the set of scenarios is considered to sufficiently cover the scope for which the LP was designed (as it stands, the question of the coverage of the application scope by the set of test scenarios is up to the tester - see sections 6.4), each scenario is implemented then executed, and the results analyzed. The metrics previously defined support measures that enable testers to evaluate the model's behavior and determine whether it satisfies $P2$, i.e., whether the test passes or not.

It is then possible to draw general conclusions about the learning program depending on possibly inadequate model behaviors that have been highlighted in the test. Note that the issues related to defect location and their correction are beyond the scope of this paper.

This process (represented in Figure 5) can be repeated for different formulas, answering so to **RQ2**. Besides, since testing online LP is costly, it is crucial to automate the process as much as possible. In the following section, we propose an answer to **RQ3**, which covers the *Implementation*, *Run* and, partially, *Analysis* phases of the testing process.

5 TOOLING

This section first presents the general principles of scenario implementation and automatic execution. Then, it describes the application of these principles to the testing of OCE's learning program, supported by two tools: Maker and Runner.

5.1 General Principles

To achieve the scenario-based approach and provide abstractions and facilities to testers, several tools are required. To implement a scenario, domain experts need a scenario editor. To describe the learning data, the editor must enable the expression of the learning interactions, each containing a learning situation with its associated ideal output. In addition, it must allow evaluation interactions to be expressed, i.e., situations in which the model must produce a result, as well as the measurement functions that evaluate the expected properties of the results. Finally, the tool must translate the scenario into an automatically executable form.

To automatically run a scenario with the LP under test, a second tool is required. This one orchestrates the execution of a scenario by sequentially running the interactions. It interacts with the online LP under test, providing both learning situations and feedback. In an evaluation interaction, the tool also provides a situation, gets the LP output, applies the measurement functions to it and delivers the measured values.

5.2 Application to OCE

To apply these principles to the test of OCE's learning program, we have developed two tools: OCE Scenario Maker to edit OCE scenarios and OCE Scenario Runner to automatically run them. A short additional video[2] demonstrates their usage in the scenario described in Section 4.1.

5.2.1 OCE Scenario Maker

This is an interactive tool for implementing OCE scenarios. In practice, the tester can reuse or define fictitious components (Figure 6, left panel). For a learning cycle, according to the formula they want to test, they can drag and drop components into a panel to define the ambient environment, and bind their services to define the ideal assembly, following *P*1 (Figure 6, right panel). The same principle applies for evaluation cycles. In this case, the tester sets a measurement function to assess the satisfaction of the property they expect on the assembly proposed by OCE (*P*2). For example, the function can measure a distance (e.g., a Jaccard Similarity Index) between the proposed assembly and an expected one, or an average distance with several assemblies, or the presence of a specific component or connection in the proposed assembly. Additional features, such as the ability to duplicate a cycle, reduce the tester's workload. From a sequence of cycles, Maker generates a JSON[3] file that implements a scenario intended to be automatically run with the second tool, Runner.

5.2.2 OCE Scenario Runner

This is a Java application that, coupled with OCE, allows the automated execution of OCE scenarios in JSON format, such as those generated by Maker. The tester selects a scenario to run. Various parameters need to be set, such as learning hyperparameters (e.g., exploration rate of reinforcement learning), and the version of OCE to test. To mitigate the impact of the non-deterministic nature of machine learning, Runner allows to repeat the execution of the scenario.

Once the parameter values are set, it operates coupled with OCE without further intervention from the tester. To assess the relevance of OCE's outputs, Runner runs the measurement function at each evaluation cycle, then provides an average value over all evaluation cycles, last an average value over all scenario repetitions. This measure indicates whether the constructed models made relevant propositions based on what OCE's learning program learned, i.e., based on the user's preferences.

[2]https://www.irit.fr/OppoCompo/makerrunnerusecase2024/

[3]https://www.json.org/json-en.html

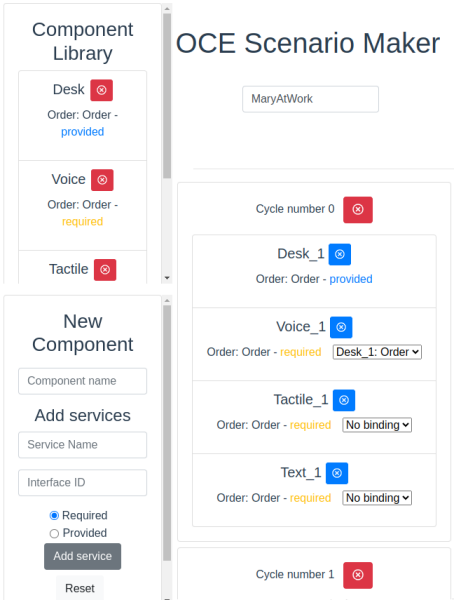

Figure 6: Maker's interface for scenario implementation.

5.2.3 Architecture

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 (OMG, 2017). Figure 7 represents 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 (*ProcessOneEnv* service). **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 (*Feedback* service).

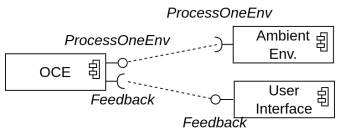

Figure 7: Architectural view of OCE in production configuration.

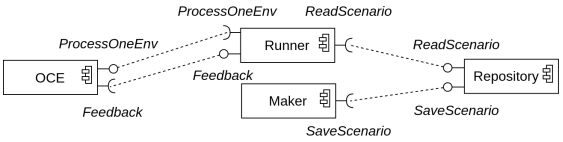

Figure 8: Architectural view of OCE in testing configuration.

Figure 8 shows how Maker and Runner have been

integrated into OCE architecture. The modularity of this architecture allows replacing **Ambient Env.** and **User Interface** by **Runner** to perform tests implemented with **Maker**. In the testing configuration, **Runner** executes a scenario taken from the **Repository** of scenarios (*ReadScenario* service). The directory is populated by scenarios implemented with **Maker** (**SaveScenario** service). For each cycle of the scenario, **Runner** requests **OCE** to propose an assembly (*ProcessOneEnv* service). In learning cycles, following the proposition, **Runner** provides **OCE** with the ideal assembly (*Feedback* service), which **OCE** treats as user feedback. In evaluation cycles, **Runner** computes quality measurements on the assemblies proposed by OCE.

6 VALIDATION

We have applied the scenario-based testing approach to conduct a test campaign of OCE's learning program. The aim of this campaign is to evaluate its ability to learn user preferences in various use cases. We present three of them, all related to the disappearance and appearance of components that are involved in the user's favorite assemblies. More specifically, the testing process described in Section 4.2 is applied to:

1. The disappearance and reappearance of a component involved in the user's favorite assembly.
2. The disappearance and appearance of two components used together and preferred by the user.
3. The disappearance and appearance of components in a realistic case of a smart home.

For each use case, i.e., for each formula to be evaluated, several scenario were designed but only one is presented in this paper. The experimental conditions are as follows. We take the standard version of OCE with default settings. In these settings, the value of the reinforcement learning exploration coefficient is 0.1. This means that the OCE's model has a 10% probability of exploring alternative connections between components rather than exploiting knowledge acquired about user preferences. In addition, to smooth out the impact of random factors, each scenario is run 100 times.

6.1 Use Case 1: Disappearance and Reappearance of a Component

Here is the first use case for the scenario-based testing approach: as OCE testers, we want to verify that **IF** *the user expresses a preference for a component in their ambient environment and this component disappears* ($P1$), **THEN** OCE *proposes again an assembly including this component when it reappears* ($P2$). Different scenarios were designed for this use case (by varying the number of cycles, appearances and disappearances...) using two tactics:

- Limit the complexity of the $P1$-compliant training sequence, i.e., the number of components, services, and cycles.
- Ensure that OCE's model has to make a choice between several possible connections at each cycle (decide), in order to leverage knowledge of user's preferences and make them evolve from user's feedback.

In order to evaluate $P2$, the tester chooses to use the Jaccard Similarity Index between the assembly proposed by OCE and a given expected one.

A scenario designed for this use case is the three-cycle scenario described in Section 4.1: the component **Voice**, preferred by Mary in Cycle 0, disappears in Cycle 1 then reappears in Cycle 2. Cycles 0 and 1 are the training data (D) satisfying $P1$. The goal is to verify that OCE's model proposes again the assembly **Voice-Desk** when **Voice** reappears, i.e., $P2(M_D)$.

This scenario was implemented using Maker and run using Runner. Running showed that out of the 100 runs, in about 90 the similarity index calculated between the assembly proposed by OCE and the expected one in the evaluation cycle equals 1.0, indicating that the produced and expected outputs are identical. This means that OCE did actually propose the assembly with the component **Voice** when it reappeared. In the other 10 runs, OCE did not propose this assembly, due to exploration during the evaluation cycle. The other designed scenarios showed the same results. Thus, for this use case, testing did not reveal a value of D such as formula 2 is true, i.e., no incorrect behavior of OCE's learning program was identified.

In the following use cases, test design was based on the same tactics and distance measurement function. So, we do not repeat these points in the text.

6.2 Use Case 2: Disappearance and Reappearance of Two Components Used Together

In this use case, we take component disappearance and reappearance a step further, where two components used together are affected. This time, we want to verify that **IF** *the user expresses a preference for using two components together* ($P1$), **THEN** *OCE later selects these two components when they are available together again* ($P2$).

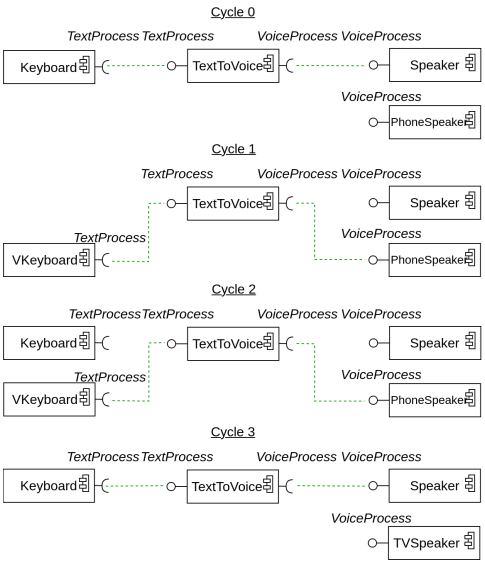

Figure 9: Learning cycles of the use case 2.

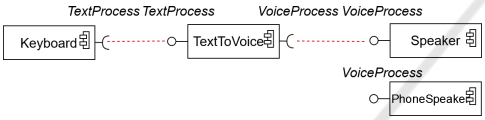

Figure 10: Evaluation cycle of the use case 2.

Here is a scenario with components for a text-to-speech application. It includes four learning cycles that define D, which satisfies $P1$ (Figure 9) followed by one evaluation cycle that specifies $P2$ (Figure 10). When the user uses the keyboard (**Keyboard**), they prefer to use the room's speaker (**Speaker**) at the same time. This preference is expressed by the ideal assemblies of cycles 0 and 3. But when the smartphone's virtual keyboard (**VKeyboard**) is available in the ambient environment, they prefer to use it together with the smartphone's speaker (**PhoneSpeaker**). This is expressed by the ideal assemblies of cycles 1 and 2. In this context, we would like to check that if **VKeyboard** disappears, OCE actually proposes to use **Speaker** together with **Keyboard**.

This scenario yielded some interesting results. In about 90 runs, the similarity index is equal to 0.33. This score means that the produced output is far from the expected one. In these runs, OCE proposed an assembly with **PhoneSpeaker** even though **VKeyboard** was unavailable, thus did not learn well the user's preferences. In the other 10 runs, OCE did propose the expected assembly due to exploration at cycle 4. This result shows a value of D for which the formula 2 is true. This use case thus reveals an unexpected behavior of OCE's learning program, probably requiring further tests and in-depth analysis which could lead to the conclusion that, in such cases, OCE fails to generalize what it has learned.

6.3 Use Case 3: Towards a More Realistic Use Case

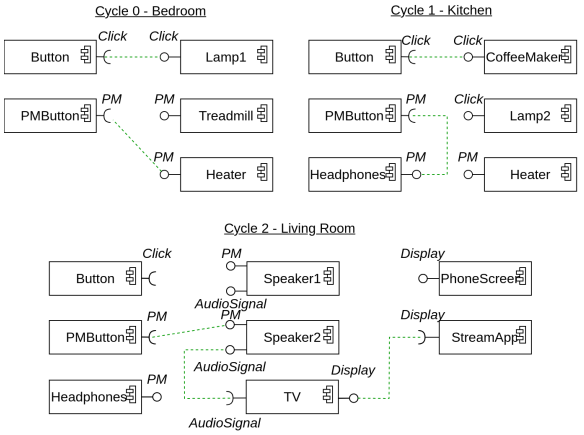

Figure 11: Learning cycles of the use case 3.

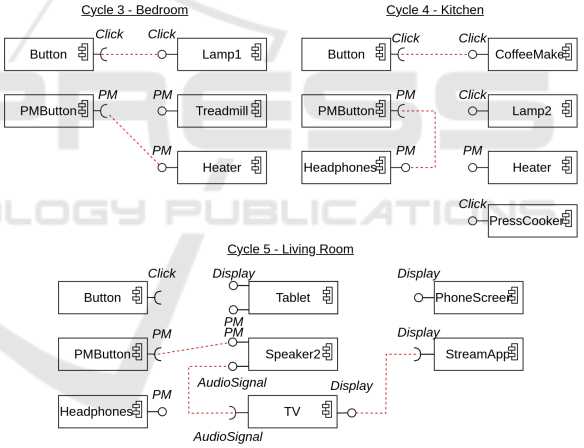

Figure 12: Evaluation cycles of the use case 3.

In this use case, we want to test the behavior of OCE's learning program in a more realistic context of a smart home. The aim is to verify that **IF** *the user moves around several rooms in their home, involving a series of appearances and disappearances of components in their surrounding environment (P1)* **THEN** *when the user returns to these rooms, OCE proposes their favorite assemblies in the room, despite slight variations in the ambient environment (P2).* To do so, we designed a scenario where the user navigates between the bedroom, kitchen and living room. As a result, several components become available or unavailable in their ambient environment depending on the room. Besides, the user has a smartphone pro-

viding a button for switching a device on or off, and a plus-minus button for adjusting a device (e.g., the volume of a speaker). Embedded in the user's phone, which they keep close at all times, these two components are permanently available.

The scenario comprises three learning cycles (Figure 11) followed by three evaluation cycles (Figure 12). When the user wakes up, they are in their bedroom. Here, there are three components: a heater, a treadmill and a lamp. In this context, the user prefers to use the first button provided by the smartphone to turn on the lamp, and the plus-minus button to turn down the heating. In the cycle 0 of the scenario (see Figure 11), these preferences are translated into the ideal assembly. Next, the user heads for the kitchen to make a coffee. On the way, they find their headphones and want to listen to the news on the radio. The user then uses the buttons on their smartphone to switch on the coffee machine and adjust the volume of the headphones, which corresponds to the ideal assembly in the cycle 1. Finally, the user sits down in the living room to watch an episode of their favorite show. To do this, the user uses their smartphone's streaming application to project the show onto the TV, using the second speaker (ideal assembly of the cycle 2).

This sequence is repeated the next day. In the scenario, it is defined by the three evaluation cycles, to check that OCE's learning program learned the user's preferences in each room of the home. However, a few changes took place: the user has received their brand-new pressure cooker, which they store in the kitchen. This new component can be turned on/off via the button of the smartphone. Besides, in the living room, the battery of their tablet is fully recharged. The user might now project shows on it and adjust the volume using the plus-minus button. Lastly, the first speaker in the living room is broken thus is no longer available in the ambient environment. These changes have an impact on the composition of ambient environments of the evaluation cycles, but as the user preferences remain unchanged, the expected assemblies are the same as the ideal ones that have been stated in the learning phase.

For each evaluation cycle, in about 90% of runs, we got an average similarity score equals to 1.0, which means that OCE's model proposed the right assembly. In the remaining 10%, the score is lowered (between 0.33 and 0.5) because OCE's model explored one or several connections in its proposition. Here, as for the use case 1, testing did not reveal an incorrect behavior of OCE's learning program.

6.4 Lessons Learned

The scenario-based testing approach has been applied beyond the three use cases described in this paper. It made possible to test both simple and more complex cases, with a greater number of cycles, components and services. The test campaign highlighted situations where OCE's LP behaves as expected, but also the presence of defects, e.g., in the way the user feedback was taken into account by the learning program. Despite its black-box nature, making default location and correction difficult, the revealed defects have been corrected by the OCE development team.

At this stage of our work, our solution suffers from certain limitations. In particular, domain experts have to design test cases by hand. In addition to the formulas to be tested, they must imagine the scenarios they believe to be the most significant, which can easily lead to oversights. As it stands, designing a scenario-based test set (see Fig. 5) is rather cumbersome and costly, without guaranteeing a good coverage of the application domain. This point argues for more automation within the process to further answer **RQ1**.

7 RELATED WORK

7.1 Testing ML-Based Systems

Research work on testing and formal verification of AI-based software systems is reviewed in (De Angelis et al., 2023). Research on testing software based on ML is relatively recent compared to work on software testing in general. Several papers address the general problem of developing ML-based software from a software engineering perspective but provide only brief explanations on testing issues (Giray, 2021; Martínez-Fernández et al., 2022).

However, significant papers have been published in recent years. Zhang et al. (Zhang et al., 2020) provided a comprehensive study on testing ML-based software, focusing on model evaluation and properties such as accuracy, robustness, and fairness. Among other things, a testing process is defined which targets the ML model and debugging: it is organized in two phases, offline testing (before deployment) and online testing (after deployment, in a real environment). Our process is limited to the offline phase and focuses on the learning program. In (Braiek and Khomh, 2020), Ben Braiek and Khomh reviewed the engineering issues of ML programs and examined the testing practices. Unlike our point of view, which separates the learning part (the LP) and the decision-making part (the model), models are considered as a whole (in-

cluding the learning part) for testing, which does not highlight the problem of transforming training data into decision-making models. Riccio et al. (Riccio et al., 2020) systematically analyzed the literature and highlighted the main challenges related to testing, including test case specification, adequacy criteria, cost, and the oracle problem. In (Mazouni et al., 2023), the recent literature is surveyed but restricted to the testing of software based on neural networks.

Among the solutions for testing ML-based systems, metamorphic testing (Chen et al., 2020) is used to verify model behavior in the absence of an oracle. Our approach tackles the oracle problem but in a different way from metamorphic testing, by expressing $P2$ with a modulable level of abstraction relative to the expected assembly.

There are works that target model testing with particular concerns. The one by Mazouni *et al.* aims to reveal, not the maximum number of defects, but their diverse natures in the context of reinforcement learning (RL) (Mazouni et al., 2024). Biagiola and Tonella focus on deep RL and the test of models, called agents, by means of surrogate models (Biagiola and Tonella, 2024): for that, they propose to generate environmental configurations likely to be at the origin of agent failures. In a previous work, the same authors examined the problem of plasticity of solutions based on deep RL, i.e., their ability to adapt to environmental conditions that deviate from the training one (Biagiola and Tonella, 2022), in the context of continuous learning. Their approach allows to get measures to verify the continuous evolution of models (including their non-regression) and thus to characterize favorable and unfavorable use cases in open environments. OCE also learns continuously in open environments, but this objective differs from our own, which is to check the alignment between the training data and the model built by the learning program.

7.2 Scenario-Based Approaches

C. Kaner (Cem Kaner, 2013) defines a scenario as a story of a person trying to accomplish something with the tested product. Hussain et al. (Hussain et al., 2015) 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. 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. (Ulbrich et al., 2015) 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 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 (system level) testing. This makes it particularly suitable for evaluating machine learning-based software due to their black-box nature. We have tailored this approach for online learning by interleaving learning and evaluation phases within a scenario.

7.3 Tools

Tools like Gymnasium (Brockman et al., 2016), DotRL (Papis and Wawrzyński, 2013), and Cogment (AI Redefined et al., 2021) 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 scenario. They only offer benchmarks to compare algorithms, providing the tester raw result data that need to be analyzed. In contrast, Runner provides facilities such as tester-specified quality measures of outputs, thereby making analysis easier.

8 CONCLUSION

In this paper, we define a scenario-based approach for functional testing of online learning programs, i.e., of programs which iteratively build and update ML models. Using test scenarios designed by domain experts without ML knowledge, this approach makes them able to show how well the learning program works, identify favorable or unfavorable use cases, compare several versions or setting, or reveal the presence of defects. A testing process accompanying the approach is also defined, where the activities of implementation, execution and result analysis can be automated, at least in part. This solution has been applied to OCE, a ML-based solution that learns online human user preferences in an ambient environment. It helped to better delimit the application scope of OCE,

and revealed several defects. Although the scenario-based testing approach has only been experimented with OCE using OCE-specific tooling, we believe that the general principles, which are independent of OCE and the application domain, are transferable to other online learning systems.

REFERENCES

AI Redefined, Gottipati, S. K., Kurandwad, S., Mars, C., Szriftgiser, G., and Chabot, F. (2021). Cogment: Open Source Framework For Distributed Multi-actor Training, Deployment & Operations. *CoRR*, abs/2106.11345.

Berrar, D. (2019). Cross-validation. In Ranganathan, S., Gribskov, M., Nakai, K., and Schönbach, C., editors, *Encyclopedia of Bioinformatics and Computational Biology*, pages 542–545. Academic Press, Oxford.

Biagiola, M. and Tonella, P. (2022). Testing the plasticity of reinforcement learning-based systems. *ACM Trans. Softw. Eng. Methodol.*, 31(4).

Biagiola, M. and Tonella, P. (2024). Testing of deep reinforcement learning agents with surrogate models. *ACM Trans. Softw. Eng. Methodol.*, 33(3).

Braiek, H. B. and Khomh, F. (2020). On testing machine learning programs. *Journal of Systems and Software*, 164:110542.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym. *CoRR*.

Cem Kaner, J. (2013). An introduction to scenario testing. *Florida Institute of Technology, Melbourne*, pages 1–13.

Chen, T. Y., Cheung, S. C., and Yiu, S. M. (2020). Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*.

De Angelis, E., De Angelis, G., and Proietti, M. (2023). A classification study on testing and verification of ai-based systems. In *2023 IEEE Int. Conf. On Artificial Intelligence Testing (AITest)*, pages 1–8.

Dunne, R., Morris, T., and Harper, S. (2021). A survey of ambient intelligence. *ACM Computing Surveys (CSUR)*, 54(4):1–27.

Giray, G. (2021). A software engineering perspective on engineering machine learning systems: State of the art and challenges. *J. of Systems and Software*, 180:111031.

Hussain, A., Nadeem, A., and Ikram, M. T. (2015). Review on formalizing use cases and scenarios: Scenario based testing. In *2015 Int. Conf. on Emerging Technologies (ICET)*, pages 1–6. IEEE.

Khomh, F., Adams, B., Cheng, J., Fokaefs, M., and Antoniol, G. (2018). Software engineering for machine-learning applications: The road ahead. *IEEE Software*, 35(5):81–84.

Martínez-Fernández, S., Bogner, J., Franch, X., Oriol, M., Siebert, J., Trendowicz, A., Vollmer, A.-M., and Wagner, S. (2022). Software engineering for AI-based systems: a survey. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 31(2):1–59.

Mazouni, Q., Spieker, H., Gotlieb, A., and Acher, M. (2023). A review of validation and verification of neural network-based policies for sequential decision making. https://arxiv.org/abs/2312.09680.

Mazouni, Q., Spieker, H., Gotlieb, A., and Acher, M. (2024). Testing for fault diversity in reinforcement learning. page 136–146, New York, NY, USA. ACM.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, New York.

Murphy, C., Kaiser, G. E., and Arias, M. (2007). An approach to software testing of machine learning applications. In *Int. Conf. on Software Engineering and Knowledge Engineering*.

Nakajima, S. (2017). Generalized Oracle for Testing Machine Learning Computer Programs. In *Software Engineering and Formal Methods - SEFM 2017*, volume 10729 of *LNCS*, pages 174–179. Springer.

OMG (2017). *Unified Modeling Language*, chapter 11.6. https://www.omg.org/spec/UML/2.5.1/PDF.

Papis, B. and Wawrzyński, P. (2013). dotRL: A platform for rapid Reinforcement Learning methods development and validation. In *2013 Fed. Conf. on Computer Science and Information Systems (FEDCSIS)*, pages 129–136. IEEE.

Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., and Tonella, P. (2020). Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25:5193–5254.

Russell, S. J. and Norvig, P. (2010). *Artificial intelligence: A Modern Approach*. Pearson Education, Inc.

Sommerville, I. (2016). Component-based software engineering. In *Software Engineering*, pages 464–489. Pearson Education, 10^{th} edition.

Sugali, K. (2021). Software testing: Issues and challenges of artificial intelligence & machine learning. *Int. J. of Artificial Intelligence & Applications*, 12(1):101–112.

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.

Ulbrich, S., Menzel, T., Reschka, A., Schuldt, F., and Maurer, M. (2015). Defining and substantiating the terms scene, situation, and scenario for automated driving. In *IEEE 18th Int. Conf. on intelligent transportation systems*, pages 982–988. IEEE.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, 25(4):465–470.

Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2020). Agent-mediated application emergence through reinforcement learning from user feedback. In *29th IEEE Int. Conf. on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 3–8. IEEE Press.

Zhang, J. M., Harman, M., Ma, L., and Liu, Y. (2020). Machine learning testing: Survey, landscapes and horizons. *IEEE Trans. on Software Eng.*, 48(1):1–36.