Synthetic End-User Testing:
Modeling Realistic Agents Based on Behavioral Examples

Pasquale Salza
University of Zurich
Switzerland
salza@ifi.uzh.ch

Marco Edoardo Palma
University of Zurich
Switzerland
marcoepalma@ifi.uzh.ch

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

ABSTRACT

For software interacting directly with real-world end-users, it is common practice to script scenario tests validating the system's compliance with a number of its features. However, these do not accommodate the replication of the type of end-user activity to which the system is required to respond in a live instance. It is especially true as compliance might also break in scenarios of interactions with external events or processes, such as other users. State-of-the-art approaches aim at inducing the software into runtime errors by generating tests that maximize some target metrics, such as code coverage. As a result, they suffer from targeting an infinitely large search space, are severely limited in recognizing error states that do not result in runtime errors, and the test cases they generate are often challenging to interpret. Other forms of testing, such as *Record-Replay*, instead fail to capture the end-users' decision-making process, hence producing largely scripted test scenarios. Therefore, it is impossible to test a software's compliance with unknown but otherwise plausible states. This paper introduces "Synthetic End-User Testing," a novel testing strategy for complex systems in which real-world users are synthesized into reusable agents and employed to test and validate the software in a simulation environment. Hence, it discusses how end-user behavioral examples can be obtained and used to create agents that operate the target software in a reduced search space of likely action sequences. The notion of action expectation, which allows agents to assert the learned compliance of the system, is also introduced. Finally, a prototype asserting the feasibility of such a strategy is presented.

CCS CONCEPTS

• **Computing methodologies** → **Agent / discrete models**; *Neural networks*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Multi-agent systems, agents, testing, neural networks, Markov models, deep learning
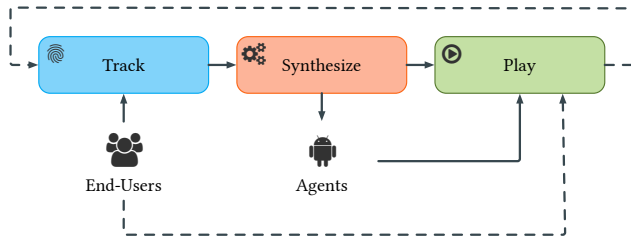
1 INTRODUCTION

Nowadays, highly popular and complex systems that enable the interaction between multiple users, e.g., social platforms, pose a unique and non-trivial challenge in software testing [1]. Such systems might be affected by issues that only arise under real-world interactions between multiple users, such as problems of security, integrity, and privacy [1]. For this reason, today's testing methods might appear limited. Traditional testing strategies overpower the developer's role, who is expected to know the popular use patterns

of the system's users and needs to compose tests for all such scenarios. Instead, state-of-the-art testing approaches, e.g., *Search-Based Software Testing*, generally optimize on code metrics, e.g., coverage, which lack correlation with actual end-user behavior [13].

One way of testing would be to use real end-users. Indeed, the system is likely to be already running in another production instance, with a community of real users interacting with it and each other. Alternatively, the platform may still be at a development status for which it would at least be possible to test it with a restricted group of testers. On the one hand, it is risky to mix testing activities with production systems. On the other hand, using human testers in an isolated development environment can hardly resemble a realistic system load. Therefore, it is paramount that the systems are tested with a simulated user activity that resembles sequences of user requests found in production. This raises two main concerns: (1) how can a simulation environment be constructed to execute such tests? (2) how can realistic user activity be created and tested?

Existing approaches have been focusing on recreating a model of the target platform, on which a simulation can be run [11]. However, generally, platforms are complex to the point that manually compiling accurate models of user behavior becomes a tedious task and intractable, with results far from the expected use patterns experienced in the target live instance. Therefore, such simulations and models remain at a high level of abstraction and are often far from reflecting a realistic scenario of a production system. One may argue that the manual scripting of test scenarios might be avoided if current state-of-the-art approaches in the field of software test generation were adapted to operate as users. Baseline solutions in this space, such as *Monkey Testing* [9] are limited to generating user activity by triggering random actions on the client and not effective in reproducing user activity [14, 15]. However, while this would potentially test for all possible activity sequences the system may undergo in production, it also tries within a much greater activity set, most of which are highly unlikely to occur. Moreover, the results of multiple random testers operating in parallel on the system might be unpredictable and uncontrollable. Instead, advanced solutions inform the action selection process through some internally evolving logic that strives to operate the target software in ever more effective ways to discover control sequences leading to some runtime error [6, 16].

Some of these strategies also explicitly target Graphical User Interface (GUI) [12] software. However, these are in a constant evolution of an outer process focused on boosting the strategy's effectiveness in maximizing some software testing-related metrics, e.g., the coverage, or a combination of such metrics. As a result, these approaches typically lead to the generation of test scenarios that are difficult to comprehend and part of a more extensive search space that includes tests largely unaware of the behavior or

Figure 1: The workflow of the approach.

intended/expected use patterns of the software they test. Moreover, tests generated through such strategies are known to be largely ineffective in the detection of bugs [2, 18].

In light of the challenges raised in this space, this work introduces *Synthetic End-User Testing*, a novel strategy for the automated testing of complex systems, in which real-world end-user are simulated by agents [20] that operate on an instance of the target system. The test scenario search space can be reduced to one consisting of the set of use patterns that users of the system in production have shown to perform. It is a desirable proposition, as it will validate a system's compliance to the set of most likely use cases, redirecting development time towards changes improving real-world use cases. Intuitively, the information for exploring this search space can be inferred by the systems' current user base. For the average case, there exists the opportunity to track both the actions and interactions of real users and how the state of the target platform changes over time. Therefore, in this context, generating test scenarios moves the focus away from generators that maximize the test coverage but instead towards minimizing the distance of synthetic end-user models to their respective and unknown end-user processes. Furthermore, current system compliance represents valuable information for encoding user expectations following some action. Modeling user expectations of the system's behavior would also allow for the statistical assertion of the system operation.

The proposed approach consists of three components, i.e., track, synthesize, and play. They are in charge of registering end-user activities, then encode into automated agents, and finally reproducing actions into a simulated environment. Each component opens to many strategies, technologies, and methods that represent the primary goal envisioned in this paper. The tracking might be implemented for different programming languages and frameworks. The synthesis of agents might be based on statistical methods or using neural networks trained on the mole of collected user activities. Finally, the reproduction of agents can target different testing goals, such as stress testing and bug discovery.

The remainder of the paper provides an overview of the approach and its components and presents a prototype of this novel approach to support its feasibility. The replication package is published at the address https://github.com/MEPalma/SyntheticEndUserTesting-Prototype [17].

2 APPROACH

The proposed approach consists of three different components, i.e., track, synthesize, and play. Figure 1 shows the workflow of how such elements are connected and interact with the "real" and mock actors. In the vision of *Synthetic End-User Testing*, each component

is highly cohesive but weakly coupled, and different technologies and methods can be combined to reach various goals for testing. For instance, one might decide to (1) *track* end-users of an ANDROID client for a social network, (2) use statistical methods to *synthesize* automated agents, then (3) *play* the agents to stress the platform under test to discover crashes or check whether the performance decays. In the following, each of the components is discussed.

Track. The first phase consists of having real end-users *using* the target application while their actions are being tracked. The user activity is registered as a sequence of bindings of application states and user actions. The data recorded must be sufficient for modeling user behavior regarding the application's state. An agent should recognize the state the application is in and, by inference from the behavior the user had previously shown to take, sample one action out of those made available to the user. The representation of the application state should also enable the agent to validate the consequences of an action it has taken. It means that if the application had consistently promoted responses of some specific trend to the same subsequence of actions, the agent should be able to assert if the application is still compliant with that behavior. A tracking layer should also enable the agent to perform an action in a manner that promotes the application to carry out the exact computation as if a real user had taken it. Tracking has to happen transparently to the end-user without showing any decay of the user experience.

Developers have to decide what and when to track, therefore, they instrument the original program code that ideally has to remain intact. This part is envisioned to be technology-dependent since a dedicated tracker has to be specifically developed for the technology employed for the clients. If a specific programming language or framework is used, there might be the possibility of leveraging some convenient functionalities. For instance, the user interface of an ANDROID app is likely based on a widget toolkit library that manages the GUI. Furthermore, a tracking procedure might then be injected into the library without the need to touch the original codebase through technologies, e.g., Aspect-Oriented Programming (AOP). It is up to the developers to decide the level of granularity of such tracking. This choice will then be relevant during the next phase of synthesis.

Synthesize. Once track(s) of individual end-users have been recorded, it is time to model the agents. An agent is meant to be a synthetic encoding of the user behavior its target end-user has led to observe. As black boxes, agent models are given user tracking data, then acting as predictors of the following action given the application state, set of available actions, and some internal state. The leveraging of end-user activity opens the door to a number of candidate strategies for the encoding of user processes.

A baseline approach might assume that the variability of action selection for a given user is only dependent on the system's current state. Hence, user data might be compiled into baseline models such as Markov Chain [7, 10], in which the system states are nodes of the decision graph. A set of actions are sampled with the observed frequency probability, a procedure that, in turn, semantically leads the user to request the system to transition from one state to another.

Alternatively, inspiration from recent leaps in the building of language models [5, 19] may be considered for the generation of

user action models. The training of language models in Masked Language Model (MLM) and Next Sentence Prediction (NSP) [19] share many similarities with user action modeling. An initial approach may consider *masked words* to be *masked actions*, allowing the model to learn the bidirectional structure of action sequences, while the recognition of ordered entailment between sets of action sequences might promote the learning of the semantics, or simulate the context, for which two or more such sequences follow each other.

Finally, the opportunity to produce generative user action models should be considered with approaches such as Generative Adversarial Networks (GAN) [8] models. Similarly, information about the system's response, or resulting state, to user actions might be used for the modeling of user expectations. For this purpose, deep auto-encoder models used in anomaly detection may be adapted to this problem [3, 4, 21]. If trained to reconstruct the system's state following a user's action, future reconstructions may signal the level of surprise about the system's operation.

Play. The generated agents are now ready to be deployed into a testing environment. *Synthetic End-User Testing* is not necessarily tied to a specific testing goal. One might decide to test the system's performance under stress through the mean of several multiple realistic users or discover bugs that derive from the simultaneous use of the system, which are otherwise difficult to capture through traditional unit testing. The possibilities are numerous, but each of them needs to manage the reproduction of agents accordingly.

The agents can be executed in a playground where they behave like the original end-users. Reproducing exactly what happened during the tracking phase is infeasible since the concurrency implicitly causes a non-deterministic scenario. However, the approach is meant to reproduce realistic behaviors, therefore, originating the expected user interactions. The play engine has to manage the execution of the agents, who observe the reality and detect a status, then decide what actions to perform. This could be done in real-time if relevant to the testing purposes. Otherwise, the time in the reality of the playground can be accelerated. Furthermore, as during the tracking phase agents have learned to statistically infer expectations about changes in the application's state following some action, the target software may also be monitored with regard to its consistency in complying with the expected/past behavior. At the same time, the opportunity to discover runtime errors remains available.

Optionally, real end-users might take part in the interaction. Ideally, this should still happen in a dedicated playground, not a production environment. Moreover, what happens in the playground can be further tracked to generate new and possibly improved data. In this case, the room for methods and technologies is widely open, and one may choose to leverage reinforcement learning strategies.

3 PROTOTYPE

This section presents the prototype of *Synthetic End-User Testing* developed, and available in the replication package [17]. This work addresses feasibility of the approach and promotes discussions about future work. Some key aspects addressed include: (1) user tracking logics to be injected into a codebase without the latter being adapted for this purpose; (2) the recorded tracking data to be well-formed and useful for the synthesizing of user behavior; (3) an

agent to be enabled to simulate user interactions without adding explicit support for codebase without the latter being adapted for this purpose; (4) a synthesized agent to rely on past user behavior data to decide what user action to take; (5) an agent to be constructed to assert the results of an action based on past user behavior data.

3.1 System Design

The subject of this preliminary feasibility study is a small-scale replica of Twitter. It provides the users with a *GUI* client with which they can connect to a remote server and access a set of functionalities resembling those offered by the popular social platform. In particular, users can: sign-up, login, logout, view the list of other users registered, choose to follow or unfollow a user, publish new tweets, retweet to and like or unlike any of the tweets displayed, share images (generated) with their tweets and retweets, view their or users the follow's tweets, hence view direct retweets made to their tweets, view who has liked a tweet, and receive push alerts when their tweets have been liked, or a user has started following them.

The application is written in Java, and employs Java Swing for the definition of the GUI and respects the default Model-View-Controller (MVC) -based *Observer* design pattern. Furthermore, the application includes several typical GUI features such as: asynchronous operations (on most actions, e.g., tweet/retweet, like, menus), pop-up windows, both custom and default User Interface (UI) components, as well as graphically reacting on events not promoted by the user events, e.g., receiving alerts.

3.2 Tracking Layer Injection

In this prototype, a tracking layer was woven into the client application through Aspect-Oriented Programming (AOP), and required no modification of the codebase. Instead, a separate AspectJ project was developed. The tracking layer is in charge of intercepting and logging user actions, injecting hooks for the programmatic triggering of UI events, and maintaining an internal representation of the GUI's current state and available actions.

It does so by first intercepting declarations of UI components to which user action events are added. Once one of such components is created, the layer appends extra logic that allows it to be notified whenever this becomes the subject of a user action. Upon receiving such a notification, the layer logs the event and its respective date and application state. A weak pointer to the components is kept and referenced by an *id*. Such *id* represents the layout location of the component in the UI, as the ordered nesting sequence of its parents, and the text the component carried (when available), thereby making it formal and session-independent. A numbering system handles the instances in which components list multiple children. As a result of this naming convention, active components, i.e., currently on screen, are those that are bounded to an *id* starting with the application's window. Whenever structural UI changes occur, these are intercepted, and the *id*s of the action-components are recomputed. Therefore, the tracker maintains an up-to-date representation of the component carrying actions the user may currently choose to take. In turn, this allows for the programmatic triggering of user events. In fact, given the *id* of a target component

and the description of a user event (e.g., click), an AWT event is created and invoked on the component.

Finally, for this first prototype, the application state was represented as the name of the view, or page, the user is currently viewing upon taking action. Declarations and updates of such views are handled similarly to the action components' *id*s.

3.3 Agents Syntesis

The concept of *agent* was abstracted to a finite state automata, and later three baseline agent types were derived: *ReplayAgent, RandomAgent, FrequencyAgent*. In this particular instance, an agent would: (1) (*Select*) select a user action given the current application state, the set of available actions, and some agent-specific internal logic; (2) (*Perform*) programmatically trigger the action selected; (3) (*Await*) obtain the application's state following the performing of the action; (4) (*Assert*) verify the compliance of the state based on some internal expectation; (5) return to 1.

ReplayAgent. A *ReplayAgent* is the simplest form of agent. It re-executes the same sequence of actions that some real-world user previously performed, supplied to it in the form of a log file. Therefore, action selection consists in choosing the next action in the sequence. Similarly, the state's assertion verifies that the application reached the same condition it did when the user initially performed the same action. While this design will ensure the agent reproduces realistic action sequences, these are fixed and fails to synthesize user behavior, instead of providing similar value propositions of unit/scenario tests. It not only leads to practical limitations (e.g., unavailable actions, cyclic operations) but also fails to explore action subsequences that might occur in response to external changes to the application state (e.g., interactions with other agents).

RandomAgent. A *RandomAgent* is another baseline agent design. It consists of an agent that selects one of the available actions randomly. Although it is eventually going to explore more use patterns than a *ReplayAgent*, in its simplest form, it is unable to construct an expectation for the application state following a random action. Hence, this design also fails to synthesize user behavior and can only unveil implementation level bugs, e.g., unhandled exceptions and error codes. This strategy seems to diverge from the goals of this project and instead results in an exploration testing strategy with a complete lack of optimization strategy.

FrequencyAgent. Finally, a *FrequencyAgent* is the only agent introduced in this prototype that makes a baseline attempt to synthesize user behavior. As the name suggests, this agent assumes the probability of choosing a specific action is only dependent on the application state. Thus, when supplied with a user's activity log, it builds an action frequency table about each state. Likewise, it builds a state frequency table about each unique tuple of application state and action taken. While the first table enables the agent to perform a weighted sampling of an action during the *Select* phase, the second allows the agent to have a weighted expectation about the resulting application state.

3.4 Play Simulation

Despite the prototype relying on strong and baseline assumptions, it proved that it is possible to inject into a target software some custom logic for the tracking of user activity and later use this information to synthesize user behavior. The feasibility of using such agents to simulate end-user activity and therefore test the application was also addressed. In particular, *FrequencyAgent*s were trained on user activity performed on the target system discussed above, and later two types of bugs were artificially added to the codebase. The first induced the client application into failing to take the users back to the their feeds (or home) whenever they clicked on an alert of another user liking one of their tweets. This was designed to occur only after receiving 10 alerts, and aimed at asserting whether the agents were able to detect a break in user expectations occurring after some interactions among them. A second artificial bug introduced in the codebase consisted of a runtime exception being thrown programmatically with a certain probability every time the user requested to follow another user. Both bugs were eventually identified by the *FrequencyAgent*s. In the case of the first bug, it correctly reported the weighted expectation of the action following the click on the like alert to consist of the feeds page, but instead of this remaining at the alerts page. Hence, ultimately confirming that a baseline form of *Synthetic End-User Testing* was compilable for the system specification introduced in this work.

4 TAKEAWAYS

As a result of the investigations carried out in the compilation of this prototype, a number of observations were made which can inform future development in this context.

The first of these is that the data collected by the tracking layer directly affects how much the agents can learn to synthesize end-user behavior and how effective the agent can be in detecting software behavior breaking user expectations. The prototype was limited to represent the system state as the name (*id*) of the currently active page. Despite this being sufficient for detecting errors concerning the updates of pages, it makes it intractable to see any other form of errors, such as, for example, the liking of a tweet failing to update the likes counter. Therefore, future work should also record richer state representations that enable agents to make finer grain validations.

A second observation concerns the modeling of user actions. After preliminary qualitative analysis of the performances of the baseline *FrequencyAgent*, it became evident how human decision-makers do not infer action selection based on the current state alone. Instead, there appears to be an element of sub-flows in using a system, each consisting of multiple actions. These concern different tasks a user may carry out during an entire session, such as tweeting or retweeting. *FrequencyAgent* failed to effectively replicate this behavior, with actions found in those sub-flows performed by the real user with equal frequency becoming overlapped. Future efforts should therefore work towards building end-user models capturing this aspect. This may be achieved by both computing a more significant state capable of providing an opportunity to infer the actions taken earlier or adding dependency of future actions on those in the past.

As a result, future work should focus on extracting fine-grained information from the tracking phase and leverage deep learning models, such as those above described, to generate agents that can reduce the testing search space by more accurately encoding end-user processes. Efforts should also evaluate the performance trade-offs of maximizing code coverage against user behavior.

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Ralf Lämmel, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2020. WES: Agent-based User Interaction Simulation on Real Infrastructure. In *Genetic Improvement Workshop (GI)*.

[2] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 263–272.

[3] Jinwon An and Sungzoon Cho. 2015. Variational Autoencoder Based Anomaly Detection Using Reconstruction Probability. *Special Lecture on IE* 2, 1 (2015), 1–18.

[4] Laura Beggel, Michael Pfeiffer, and Bernd Bischl. 2019. Robust Anomaly Detection in Images Using Adversarial Autoencoders. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 206–222.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[6] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering (TSE)* 39, 2 (2013), 276–291.

[7] Paul A Gagniuc. 2017. *Markov Chains: From Theory to Implementation and Experimentation.* John Wiley & Sons.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. *Advances in Neural Information Processing Systems* 27 (2014).

[9] Google. 2022. *Android Monkey.* https://developer.android.com/studio/test/other-testing-tools/monkey

[10] Oliver C. Ibe. 2014. Special Random Processes. In *Fundamentals of Applied Probability and Random Processes* (second ed.). Academic Press, 369–425.

[11] Eugene Ie, Chih-wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. 2019. RecSim: A Configurable Simulation Platform for Recommender Systems. *arXiv:1909.04847 [cs.LG]* (2019). https://arxiv.org/abs/1909.04847

[12] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

[13] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[14] Mostafa Mohammed, Haipeng Cai, and Na Meng. 2019. An Empirical Comparison Between Monkey Testing and Human Testing (WIP Paper). In *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 188–192.

[15] Samad Paydar. 2020. An Empirical Study on the Effectiveness of Monkey Testing for Android Applications. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering* 44, 2 (2020), 1013–1029.

[16] Anjana Perera, Aldeida Aleti, Burak Turhan, and Marcel Boehme. 2022. An Experimental Assessment of Using Theoretical Defect Predictors to Guide Search-Based Software Testing. *IEEE Transactions on Software Engineering (TSE)* (2022).

[17] Pasquale Salza, Marco Edoardo Palma, and Harald C. Gall. 2022. *Synthetic End-User Testing: Modeling Realistic Agents Based on Behavioral Examples - Replication Package.* https://github.com/MEPalma/SyntheticEndUserTesting-Prototype

[18] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (t). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 201–211.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Conference on Neural Information Processing Systems (NIPS)*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). 5998–6008.

[20] Michael Wooldridge. 1999. Intelligent Agents. In *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* 27–73.

[21] Chong Zhou and Randy C Paffenroth. 2017. Anomaly Detection with Robust Deep Autoencoders. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. 665–674.